EQUIVALENCE PROBLEMS FOR MONADIC SCHEMAS

Joseph E. Qualitz

June 1975

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

EQUIVALENCE PROBLEMS FOR MONADIC SCHEMAS

by

Joseph Edward Qualitz

Submitted to the Department of Electrical Engineering
on May 15, 1975, in partial fulfillment of the
requirements for the degree of Doctor of Philosophy

ABSTRACT

A class of monadic program schemas is defined. This class, called iteration schemas, consists of schemas whose programs comprise assignment statements, conditional statements, and iteration statements. These schemas are shown to correspond to program schemas which are structured, and are shown to be strictly less "powerful" than the monadic program schemas.

A notion of equivalence is formalized as the functional equivalence of schemas under free interpretations, interpretations which represent symbolically the set of all interpretations of a schema. It is shown that the equivalence problem for iteration schemas is unsolvable, even if the schemas possess highly restrictive properties. Questions are raised regarding the decidability of equivalence for various subclasses of iteration schemas, and equivalence is shown to be decidable for several of these classes.

The equivalence problems for structured independent location schemas are examined in particular detail. A weak form of equivalence is shown to be undecidable for the schemas, and the general equivalence problem is shown to be related in a non-trivial manner to the equivalence problem for multi-tape finite automata.

THESIS SUPERVISOR: Jack B. Dennis

TITLE: Professor of Computer Science and Engineering

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor, Jack Dennis, for the support and advice he has provided throughout the course of my research. His help is greatly appreciated.

I would also like to thank Al Meyer for valuable suggestions offered prior to the writing of this dissertation, and Peter Elias for many helpful comments made during the preparation of this final manuscript.

Finally, I would like to thank my wife, Donna, for three years of patience, criticism, and encouragement, in just the right amounts. This work is dedicated to her.

June, 1975 Joseph E. Qualitz

This work was supported in part by the National Science Foundation under research grant GJ-34671, in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research contract number N00014-70-A-0362-0006, and in part by the IBM Funds for Research in Computer Science.

TABLE OF CONTENTS

PAGE

Abstract ---------- 2
Acknowledgements ---------- 3
Table of Contents ---------- 4
List of Figures ---------- 6
List of Theorems and Lemmas ---------- 7

CHAPTER I: INTRODUCTION 8

1.1 Schematology ---------- 8
1.2 Schematology: A Brief History ---------- 9
1.3 Schematology: Equivalence Problems ---------- 11
1.4 Outline of the Thesis ---------- 13

CHAPTER II: MONADIC PROGRAM SCHEMAS AND ITERATION SCHEMAS 16

2.1 Monadic Program Schemas ---------- 16
2.1.1 Formal Definitions ---------- 16
2.1.2 Interpretations and Executions ---------- 22
2.1.3 Free Interpretations ---------- 23
2.1.4 Classes of Monadic Program Schemas ---------- 24
2.1.5 Structured Monadic Program Schemas ---------- 26
2.2 Iteration Schemas
2.2.1 Formal Definitions ---------- 30
2.2.2 Classes of Iteration Schemas ---------- 33
2.3 Structured Monadic Program Schemas and Iteration Schemas: A Correspondence ---------- 34
2.4 Incompleteness of Iteration Schemas ---------- 37

CHAPTER III: EQUIVALENCE 41

3.1 Equivalence — Introduction ---------- 41
3.2 Strong Equivalence — Definition ---------- 46
3.3 Weak Equivalence — Definition ---------- 47

CHAPTER IV: UNSOLVABLE PROBLEMS 48

4.1 Post's Correspondence Problems ---------- 50
4.2 A Note on Notation ---------- 51
4.3 Undecidability of Equivalence for Restricted Location Iteration Schemas ---------- 52
4.4 Undecidability of Equivalence for Conditional-Free Schemas- 59
4.5 Discussion ---------- 61

CHAPTER V: SOLVABLE PROBLEMS 64

5.1 Test Sequences and Logic Equivalence ---------- 65
5.2 Productivity and Essentiality ---------- 66

PAGE

5.3 Additional Definitions and Terminology ---------- 68
5.3.1 Definitions ---------- 68
5.3.2 A Note on Terminology ---------- 69
5.4 The Elimination Theorem ---------- 70
5.5 Solvable Problems for Single Appearance Schemas ---------- 77
5.5.1 Decidability of Equivalence for Conditional-Free RLSA Schemas ---------- 77
5.5.2 Decidability of Equivalence for FRLSA Schemas ------ 89
5.6 Decidability of Equivalence for FILCF Schemas ---------- 98
5.7 Discussion ---------- 107

CHAPTER VI: SCHEMAS AND AUTOMATA 108

6.1 Multi-Tape Finite Automata ---------- 108
6.2 Equivalence Problems for Multi-Tape Finite Automata --------- 112
6.3 "Equivalence" of Independent Location Schemas and Multi-Tape Finite Automata ---------- 113
6.4 Weak Simulation of Multi-Tape Automata by Structured Independent Location Schemas ---------- 113
6.4.1 The Simulation ---------- 114
6.4.2 Weak Equivalence: A Reducibility ---------- 118
6.5 Undecidability of Weak Equivalence for Independent Location Schemas ---------- 121
6.6 Single State Automata ---------- 124
6.6.1 Single State Automata — Definitions ---------- 125
6.6.2 Strong Equivalence — A Reducibility ---------- 126
6.7 Equivalence of Multi-Tape Automata — A Reducibility Result 129
6.8 Discussion ---------- 135

CHAPTER VII: SUMMARY 136

BIBLIOGRAPHY 140

APPENDIX A ---------- 144

APPENDIX B ---------- 145

APPENDIX C ---------- 147

BIOGRAPHICAL NOTE 149

LIST OF FIGURES

PAGE

FIGURE 1: Examples of Monadic Program Schemas ---------- 17

FIGURE 2: Example of a Monadic Program Schema ---------- 18

FIGURE 3: Schemas of Figure 1 in Flowchart Form ---------- 19

FIGURE 4: Schema of Figure 2 in Flowchart Form ---------- 20

FIGURE 5: Structured Schema ---------- 28

FIGURE 6: Iteration Schema Equivalent to the Schema of Figure 5 -- 31

FIGURE 7: Non-Structured Schema and Equivalent 'While' Schema With Boolean Operators ---------- 39

FIGURE 8: Equivalent Schemas With Distinct Sets of Predicate Symbols and Function Symbols ---------- 42

FIGURE 9: Schema With Inessential Predicate Symbol ---------- 67

FIGURE 10: Summary of Results ---------- 138

LIST OF THEOREMS AND LEMMAS

RESULT	PAGE
Theorem 4.1	58
Theorem 4.2	61
Lemma 5.1	72
Theorem 5.1	74
Lemma 5.2	78
Lemma 5.3	79
Lemma 5.4	80
Lemma 5.5	81
Lemma 5.6	82
Lemma 5.7	84
Theorem 5.2	87
Lemma 5.8	89
Corollary 5.9	91
Lemma 5.10	91
Lemma 5.11	93
Corollary 5.12	95
Lemma 5.13	97
Theorem 5.3	97
Lemma 5.14	98
Lemma 5.15	102
Lemma 5.16	106
Theorem 5.4	106
Lemma 6.1	119
Theorem 6.6	120
Lemma 6.2	121
Lemmas 6.3 and 6.4	122
Theorem 6.2	123
Corollary 6.5	123
Theorem 6.3	128
Lemmas 6.6 and 6.7	130
Theorem 6.4	133

$$v \leftarrow +(x,y)$$

$$u \leftarrow v$$

$$v \leftarrow *(x,u)$$

is equivalent to the sequence of statements:

$$v \leftarrow +(x,y)$$

$$u \leftarrow +(x,y)$$

$$v \leftarrow *(x,v)$$

which in turn is equivalent to the sequence:

$$v \leftarrow *(x,x)$$

$$u \leftarrow *(x,y)$$

$$v \leftarrow +(v,u)$$

$$u \leftarrow +(x,y)$$

if we view the sequences as portions of programs expressed in some programming language in which '+' and '*' represent addition and multiplication, respectively. From a schematological point of view, however, only the first two sequences are equivalent: the last sequence will be equivalent to the others only if the function associated with the symbol '*' distributes over that associated with the symbol '+'.

1.2 Schematology: A Brief History.

The first schematized model for computation is generally attributed to Ianov [9]. In his model, the data space of a computation is treated as a single entity which is altered via a sequence of function applications, the particular function to be applied at any point in the

sequence depending on the outcomes of a number of predicates applied to the current value of the data space. Rutledge [26] later demonstrated a correspondence between the Ianov model and a class of finite state automata, and it is not surprising, therefore, that the equivalence problem for Ianov schemas is solvable.

The major difficulty with the Ianov model is that it discards much information about the essential properties of programs being represented. In particular, the data space of real computations is generally divided into a number of discrete components, and at each step in a computation functions and predicates are applied to certain subsets of these components, rather than to the total data space. The specification of the object components of each application establishes a data dependency relation among them, and it is precisely this relation which is missing from the Ianov model.

Luckham, Park, and Paterson [16] proposed a more familiar schematized model — program schemas — in which data dependency is illustrated by associating a unique symbol with each component of the data space of a computation and specifying, by means of an uninterpreted program consisting of assignment instructions and transfer instructions, the sequence of functions and predicates to be applied during the computation and the object components of each application. They were able to demonstrate that such schemas are capable of simulating in a natural way multi-head finite automata [25] and thus that the equivalence problem for program schemas is unsolvable.

Much of the subsequent work in schematology has been devoted to the study of schematized models possessing special features which facilitate discussions of computational parallelism, determinacy, productivity, etc. (cf, [3], [5], [7], [10], [14], [21], [24], [27]). In several cases, which are discussed below, equivalence problems have also been considered in the work.

1.3 Schematology: Equivalence Problems.

Equivalence problems for schematized models have been of considerable interest to theoreticians since several open problems of long standing in automata theory can be shown reducible to, or closely related to, the equivalence problems for certain classes of computation schemas. For example, it has been demonstrated that the equivalence problem for multi-tape finite automata as defined in [23] is reducible to the equivalence problem for a particularly simple class of program schemas containing only monadic function and predicate symbols; similarly, it has been shown that the equivalence problem for deterministic pushdown automata is closely related to that for the class of schemas defined by deBakker and Scott [4]. But schema equivalence problems have certain practical applications as well, particularly with regard to the design of compilers. Since compilers frequently deal with programs containing user defined functions and subroutines which have, during compilation, no semantic content, procedures for deciding whether certain program transformations are equivalence preserving in a schematological sense are apt to be of great value, particularly if program optimization or simplification is to be attempted during the compilation process. For

example, if we are interested in whether the evaluation of some externally defined function may take place during program execution at some point other than that indicated by its position within a given sequence of instructions (or indeed if it need occur at all), we are dealing with schematological issues. In fact, since program optimization generally involves changing the structure of a program without changing its functional behavior, we must either be prepared to deal with issues of schematological equivalence or restrict any optimization to those portions of a program which are completely interpreted at compilation time (or whenever such optimization is to occur). Since increasing emphasis is being placed on encouraging the widespread sharing of procedures written by many different users, each of whom can be assumed to know very little about the internal behavior of procedures written by others, the former option seems the more desirable.

Thus far results concerning the decidability of equivalence for schematized models have been quite elusive, and in most cases in which results have been obtained for fairly general classes of schemas the results have been negative (i.e. have indicated the undecidability of equivalence). The few positive results include the somewhat trivial result for Lanov schemas mentioned earlier; the results of Ashcroft Manna, and Pnueli concerning monadic functional schemas [2]; the result (for a rather strong definition of equivalence) of Keller for a class of program schemas [11]; and the results of Paterson for "progressive" program schemas (in which a computed value is immediately reused in the next computation step), and monadic program schemas

without nested loops in their control structures [18]. In any case, the equivalence problems for a great many interesting classes of schemas are open and, despite the optimistic predictions of Paterson in [19], results have not been quickly forthcoming.

1.4 Outline of the Thesis.

In this thesis, we consider the problem of deciding equivalence for classes of program schemas (as defined by Luckham et al) which have been restricted in certain ways in an attempt to make the problem somewhat tractable. Primarily, we are restricting attention to program schemas which contain only monadic function and predicate symbols and which are structured in such a way as to represent only "whilish" programs: those composed of assignment statements, conditional statements, and iteration statements (i.e. "while" and "until" statements). Our choice of program schemas as a starting point for the study is motivated by the fact that they are quite general in their ability to model computations, and they permit a high degree of informality in discussions since they are inherently familiar to anyone who has dealt with any sort of programming language. Also, the equivalence problems for certain classes of the schemas have been studied rather extensively because of their relations to well-known problems in automata theory.

We consider in this thesis several classes of schemas and our primary objective is to answer as many as possible of the questions we pose regarding the decidability of equivalence for the classes. But we are also interested in the relationships which exist among the classes

of schemas — particularly whether schemas in one class may be simulated by schemas in another, or whether the equivalence problem for one class of schemas is reducible to that for another. In this regard, the discussions in Sections 2.3 and 6.6 should be thought of not as diversions, but rather as attempts to meet this objective.

Roughly, the thesis is organized as follows:

In Chapter II we define two classes of schemas: monadic program schemas and iteration schemas. We show that the iteration schemas correspond to a class of monadic program schemas whose graph representations possess a certain topological structure, and that this class of schemas is an incomplete class of monadic program schemas.

In Chapter III we present an intuitive notion of schema equivalence, and formalize this notion as the functional equivalence of schemas for "free" interpretations, interpretations which symbolically represent families of related interpretations. We argue that this is a correct formalization of the intuitive notion.

In the fourth chapter, we demonstrate that equivalence is not decidable for a restricted class of iteration schemas and pose questions about the decidability of equivalence for classes of schemas possessing additional restrictive properties.

In Chapter V we show that the equivalence problem is solvable for certain classes of iteration schemas, and discuss the extensions of these results to more general classes of schemas.

The sixth chapter is devoted to a study of the equivalence

problems for a particular class of structured schemas (the independent location schemas) and the relation of such problems to the equivalence problems for certain automata theoretic models.

Finally, in Chapter VII we summarize the main points of the thesis and suggest areas for further study.

CHAPTER II: MONADIC PROGRAM SCHEMAS AND ITERATION SCHEMAS

In this chapter we define formally the classes of schemas we shall be concerned with, and present notions of schema interpretations and executions.

2.1 Monadic Program Schemas.

Figures 1 and 2 illustrate monadic program schemas; Figs. 3 and 4 illustrate the schemas in flowchart form.

2.1.1 Formal Definitions.

The definitions given below differ in unessential ways from those in [16] or [18].

A monadic program schema (MPS) is a five-tuple $S = (V, F, P, I, \mathcal{P})$ where:

V is a finite set of variable symbols.

F is a finite set of function symbols.

P is a finite set of predicate symbols.

I is a finite set of instruction labels.

$\mathcal{P}$ is the program of S, a finite sequence of instructions of one of the following forms:

(1) An assignment instruction of the form

$$i: x \leftarrow f(y)$$

or of the form

$$i: x \leftarrow y$$

where i is an instruction label, f is a function symbol, and x and y are variable symbols.

(a) $V = \{x,y\}$

$F = \{f,g\}$

$P = \{p,q\}$

$I = \{i_1, \ldots, i_{14}, e\}$

$\mathcal{P}$ is the program

i_1: $x \leftarrow y$

i_2: $p(x)\ i_3, i_{10}$

i_3: $x \leftarrow f(y)$

i_4: $p(x)\ i_5, e$

i_5: $y \leftarrow f(y)$

i_6: $p(y)\ i_7, e$

i_7: $x \leftarrow g(x)$

i_8: $y \leftarrow f(x)$

i_9: $p(x)\ i_2, i_2$

i_{10}: $q(x)\ i_{11}, i_{13}$

i_{11}: $x \leftarrow y$

i_{12}: $q(x)\ i_3, e$

i_{i3}: $p(x)\ i_{14}, i_{12}$

i_{14}: $x \leftarrow f(x)$

e: END

(b) $V = \{x,y\}$

$F = \{f,g\}$

$P = \{p,q\}$

$I = \{i_1, \ldots, i_{10}, e\}$

$\mathcal{P}$ is the program

i_1: $p(x)\ i_2, i_8$

i_2: $x \leftarrow f(x)$

i_3: $q(x)\ i_4, e$

i_4: $x \leftarrow g(x)$

i_5: $q(x)\ i_6, i_{10}$

i_6: $y \leftarrow g(y)$

i_7: $p(x)\ i_2, i_2$

i_8: $y \leftarrow g(y)$

i_9: $p(y)\ i_3, i_3$

i_{10}: $y \leftarrow f(y)$

e: END

FIGURE 1. Examples of Monadic Program Schemas.

$V = \{x,y,z\}$ $F = \{f,g\}$ $P = \{p,q,r\}$ $I = \{i_1, \ldots, i_{24}, e\}$

$\mathcal{P}$ is the program:

i_1: $x \leftarrow y$

i_2: $p(x)$ i_{11},i_3

i_3: $y \leftarrow g(y)$

i_4: $p(y)$ i_5,i_8

i_5: $y \leftarrow g(y)$

i_6: $z \leftarrow f(z)$

i_7: $p(z)$ e,e

i_8: $y \leftarrow f(y)$

i_9: $z \leftarrow f(z)$

i_{10}: $p(z)$ e,e

i_{11}: $x \leftarrow f(x)$

i_{12}: $q(y)$ i_{14},i_{13}

i_{13}: $q(x)$ i_{21},i_2

i_{14}: $r(y)$ i_{15},i_{18}

i_{15}: $y \leftarrow g(y)$

i_{16}: $z \leftarrow f(z)$

i_{17}: $p(z)$ i_{12},i_{12}

i_{18}: $y \leftarrow f(y)$

i_{19}: $z \leftarrow f(z)$

i_{20}: $p(z)$ i_{12},i_{12}

i_{21}: $x \leftarrow g(x)$

i_{22}: $y \leftarrow f(y)$

i_{23}: $z \leftarrow f(z)$

i_{24}: $p(z)$ i_{13},i_{13}

e: END

FIGURE 2. Example of a Monadic Program Schema.

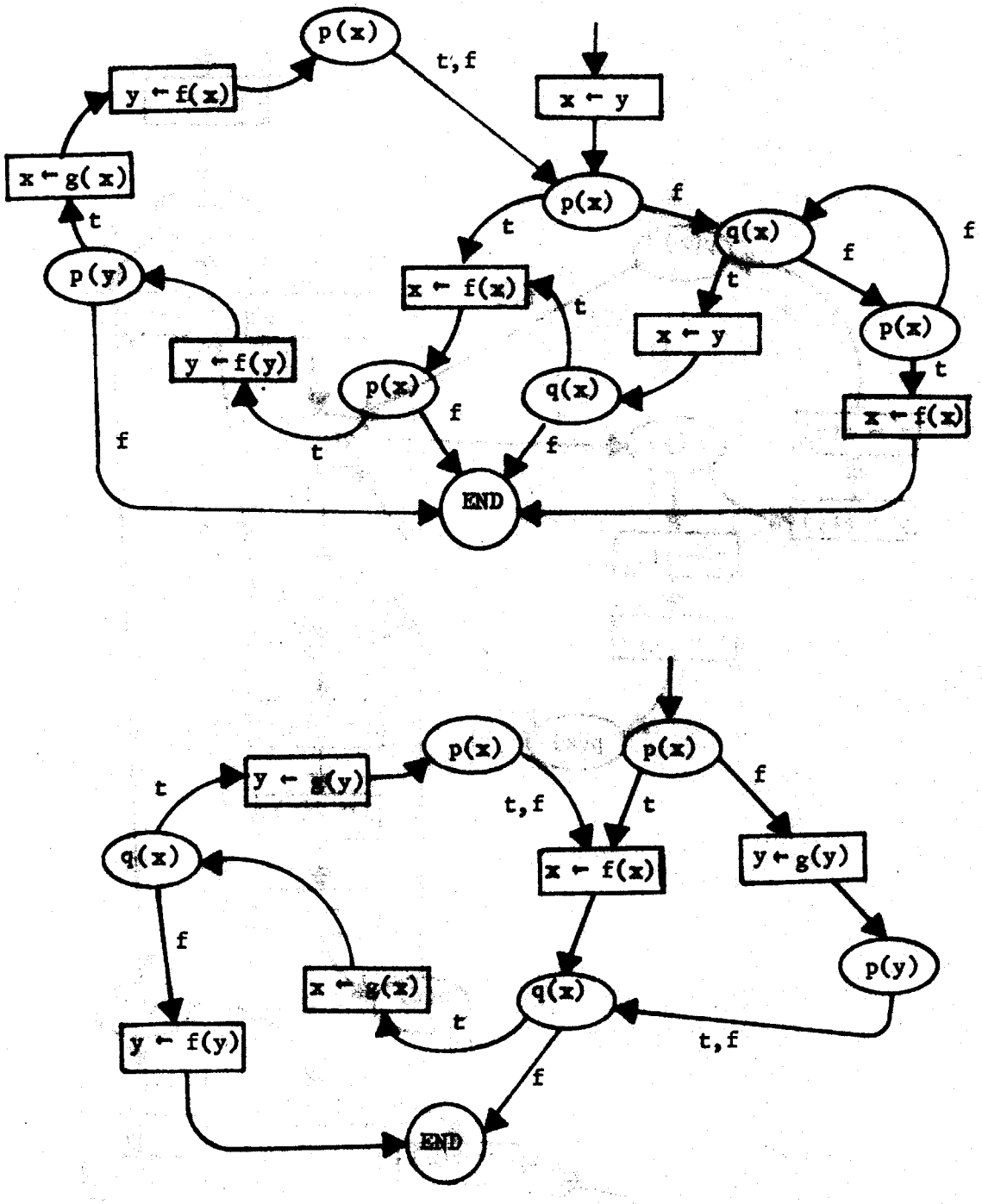


FIGURE 3. Schemas of Figure 1 in Flowchart Form.

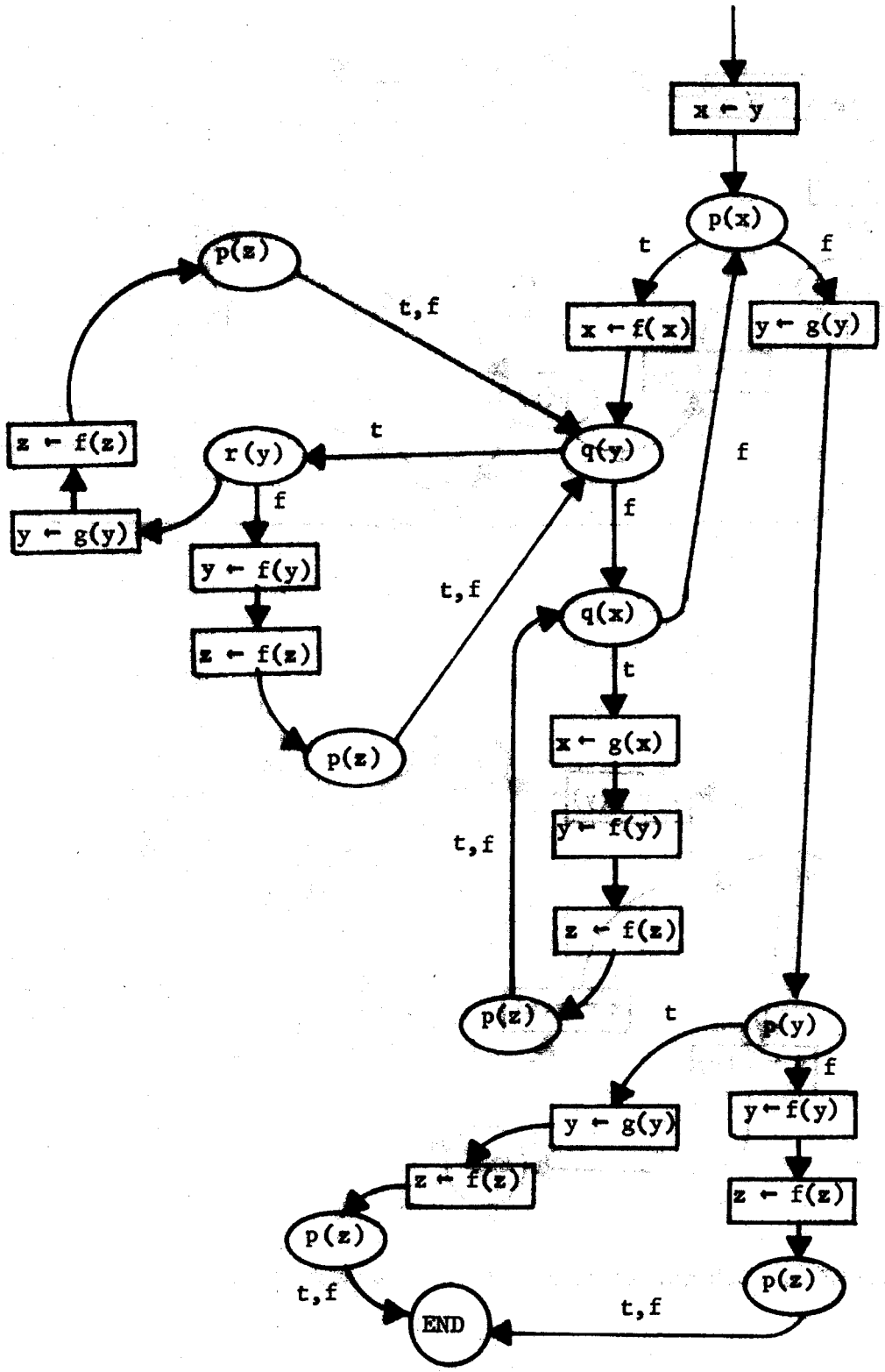


FIGURE 4. Schema of Figure 2 in Flowchart Form.

(2) A transfer instruction of the form

i: p(x) i_t, i_f

where p is a predicate symbol, x is a variable symbol, and i, i_t, and i_f are instruction labels. (We refet to i_t as the true alternative of the instruction and to i_f as the false alternative.)

(3) A terminal instruction of the form

i: END

where i is an instruction label.

We require that the program end with a terminal instruction and that this be the only such instruction in the program. In addition, we require that the label appearing as the left portion of each instruction be unique and that the true alternative and false alternative of each transfer instruction appear as the left portion of some statement in the program.

Program schemas are conveniently represented in flowchart form. A flowchart for a schema is a directed graph whose nodes represent the instructions in the program of the schema. An assignment instruction is represented in the flowchart by a rectangle containing the righthand portion of the instruction; a transfer instruction is represented by an oval containing the center portion of the instruction; a terminating instruction is represented by a circle. An unlabelled arc is drawn from a node n to a node n' in the flowchart if n represents an assignment instruction in the program of the schema and n' represents the instruction which follows in the program. An arc labelled 't' is drawn from node n

to node n' in the flowchart if n represents a transfer instruction in the program and n' represents the instruction labelled with the true alternative of the transfer instruction. An arc labelled 'f' is drawn from node n to node n' if n represents a transfer instruction and n' represents the instruction labelled with the false alternative of the transfer instruction. A single arc, emanating from no node of the flowchart, is drawn to the node representing the first instruction in the program. No other arcs appear in the flowchart.

Clearly, representing a schema by a flowchart provides us with no new information about the computation being represented. A flowchart is merely a convenient form for representing the control structure of the schem's program and facilitates discussions of certain topological features of the structure.

2.1.2 Interpretations and Executions.

We provide an interpretation for an MPS by specifying a domain D of individuals; total monadic functions from D into D to be associated with the function symbols of the schema; total monadic predicates from D into {true,false} to be associated with the predicate symbols of the schema; and particular elements of the domain to be associated initially with the variable symbols of the schema. Each such interpretation associates an idealized computer program with the schema and a corresponding execution of the schema, defined as follows:

We initially associate the appropriate elements of the domain with the variable symbols of the schema. We then execute, in the order in

which they are encountered, the instructions in the schema's program. Execution of an assignment instruction of the form 'i: $x \leftarrow f(y)$' causes the element $\varphi_f(\delta)$ to be associated with the variable symbol x, where φ_f is the function associated with f by the interpretation and δ is the element associated with symbol y at the time of the execution. Executing an assignment instruction of the form 'i: $x \leftarrow y$' causes the element associated with y at the time of the execution to be associated with x. After executing either type of assignment instruction, we proceed to the next instruction in the program.

Executing a transfer instruction of the form 'i: $p(x)\ i_t, i_f$' consists of evaluating $\Pi_p(\sigma)$, where Π_p is the predicate associated with p by the interpretation and σ is the element associated with x at the time of the execution, and then branching to the instruction labelled with i_t or i_f, according as the outcome of the evaluation is <u>true</u> or <u>false</u>.

Executing the final instruction of the program causes the execution to terminate. If an execution terminates, the final set of elements associated with the variable symbols of the schema is taken to be the <u>value</u> of the schema for that execution and for the corresponding interpretation. (The value is undefined if the execution fails to terminate.)

2.1.3 Free Interpretations.

In subsequent chapters we will deal almost exclusively with a particular subset of the possible interpretations for a schema: the

free, or Herbrand, interpretations.

Formally, a free interpretation for an MPS with variable symbols V, function symbols F, and predicate symbols P consists of:

(1) The domain $D = F^* \cdot \Delta_V$, where Δ_V is the set $\{\Delta_x \mid x \in V\}$.

(2) The initial association of the element Δ_x with each symbol $x \in V$.

(3) The association of the total function φ_f: $D \to D$ defined as $\varphi_f(\delta) = f \cdot \delta$, $\delta \in D$, with each function symbol f in F.

(4) The association, with each predicate symbol p in P, of some total predicate Π_p: $D \to \{\underline{true}, \underline{false}\}$.

We note that each free interpretation for a schema has the same domain, a set of strings which represent symbolically the values which may be associated with variable symbols during executions of the schema. In fact, the significance of the free interpretations is that they represent symbolically the set of all interpretations for a schema. In particular, we shall argue in the next chapter that, with regard to equivalence problems, we may restrict attention solely to free schema interpretations.

2.1.4 Classes of Monadic Program Schemas.

In this section we define three classes of monadic program schemas: the free schemas, the independent location schemas, and the restricted location schemas. A fourth class, the structured schemas, is discussed in the following section.

An MPS is free if no predicate is ever applied twice to the same

value during an execution of the schema defined by any free interpretation for S. Intuitively, if S is free then either outcome is possible whenever we apply a predicate to a value during an execution defined by a free interpretation for S — the outcome is not constrained by the outcomes of previous applications.

The reader may verify that the schemas of Figs. 1(b) and 2 are free, while that of Fig. 1(a) is not.

An MPS is an <u>independent location schema</u> if every assignment instruction in its program is of the form

$$i: x \leftarrow f(x)$$

for some variable symbol x and some function symbol f, i.e. if the argument variable and the assignment variable are one and the same in each assignment instruction. (If we associate a location in a data space with each variable symbol of such a schema, these locations are independent in the sense that each value computed during an execution of the schema is stored back in the location from which the corresponding argument value was obtained -- hence the name independent location.)

The schema of Fig. 1(b) is an independent location schema; the schemas of Figs. 1(a) and 2 are not independent location.

An MPS is a <u>restricted location schema</u> if it is an independent location schema except, possibly, for some number of initial assignment instructions of the form

$$i: x \leftarrow y$$

where x and y are variable symbols and i is an instruction label which

is not the true or false alternative of any transfer instruction in the schema's program. (This last condition ensures that such instructions are executed only at the start of each execution of the schema.)

The schema of Fig. 2 is a restricted location schema, as is the schema of Fig 1(b). (Clearly, any independent location schema is a restricted location schema.) The schema of Fig. 1(a) is not a restricted location schema.

We note here (and the reader may verify following Chapter III) that the restricted location schemas form an incomplete subclass of the monadic program schemas, and the independent location schemas form an incomplete subclass of the restricted location schemas. We note also that the free monadic program schemas, free restricted location schemas, and free independent location schemas form incomplete subclasses of the monadic program schemas, restricted location schemas, and independent location schemas, respectively.

2.1.5 Structured Monadic Program Schemas.

In the flowcharts of Figs. 3 and 4 there are several instances of an arc labelled with both 't' and 'f' emanating from a transfer node, an obvious shorthand notation for a pair of arcs, one labelled 't' and the other labelled 'f', which terminate at the same node. In such a case, the corresponding transfer instruction in the represented program is clearly functioning as a 'GoTo' statement, necessary in a sequential program but superflous in a flowchart representation.

We may eliminate such transfer nodes from a flowchart after redirecting any arcs which terminate at such a node to the node's successor. We refer to the resultant flowchart as reduced, and note that while it may no longer represent a valid program schema, it never-the-less represents a computation functionally equivalent to the original.

Figure 5 shows the reduced flowchart constructed from the flowchart of Fig. 4. It also shows the decomposition of the reduced flowchart into simple blocks:

A block is a piece of reduced flowchart which has no more than a single exit (which may consist of many arcs that terminate at a single node) and arbitrarily mnay entrances. A transfer node in such a block is a main node of the block if it is not contained in some sub-block, and a block is a simple block if it contains no more than one main transfer node. We say that an MPS is structured if it can be decomposed into simple blocks, as is the case in Fig.5.

A few words about the above definition are in order:

Most of the results in this thesis pertain to monadic program schemas which are structured in such a way as to represent 'while' schemas — schemas whose programs comprise assignment instructions, conditional instructions, and iteration ("while" or "until") instructions. It has been demonstrated by Ashcroft and Manna [1] that any program schema may be transformed into a schema so structured, in such a way that its output behavior is not affected; in addition, if

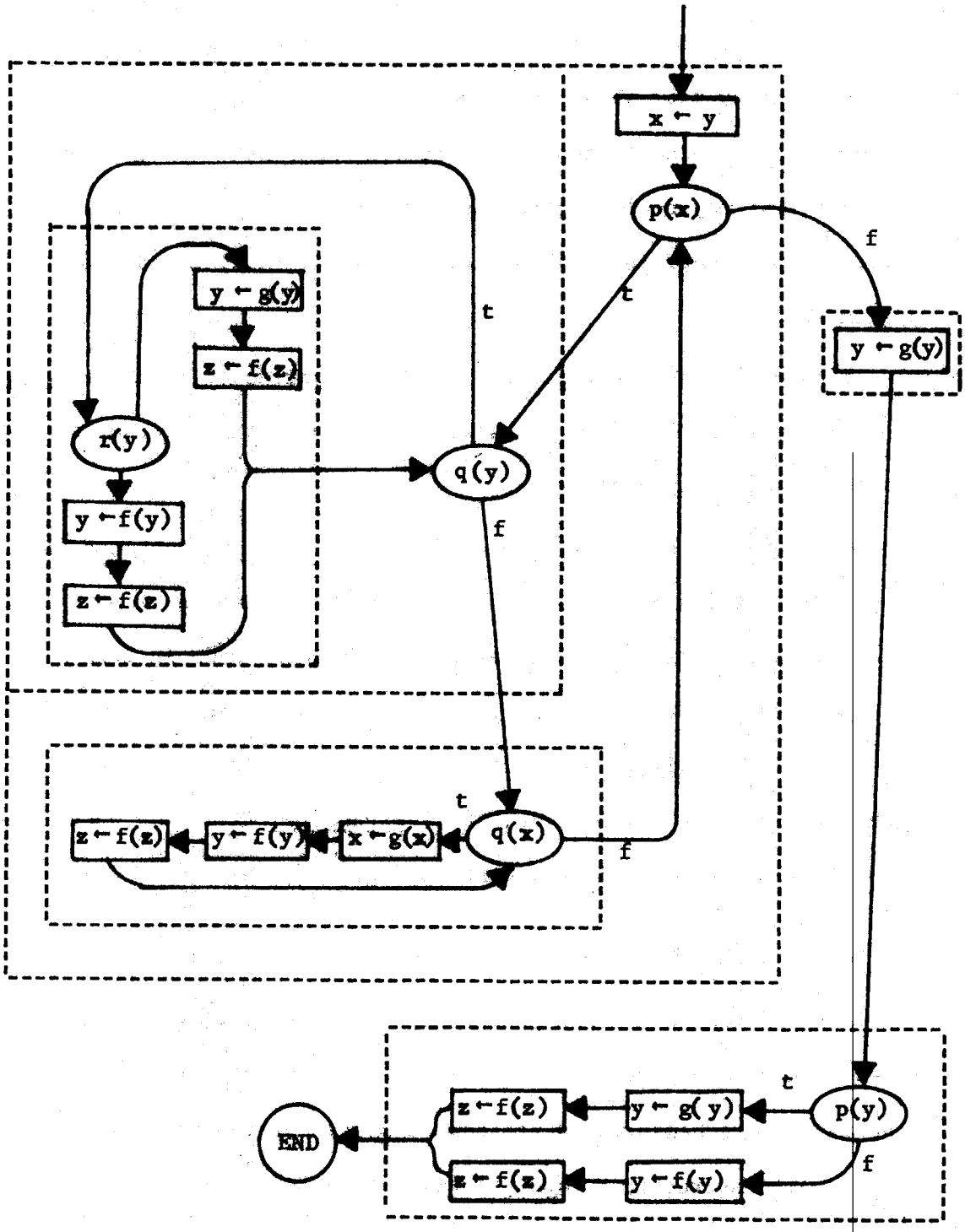


FIGURE 5. Structured Schema.

the original program schema contains only monadic function and predicate symbols, then so does the resultant structured schema.

Unfortunately, the Ashcroft and Manna results have, for our purposes, a serious drawback: the resultant 'while' schemas will contain, in general, conditional and iteration instructions containing compound predicate expressions, i.e. expressions composed of a number of simple predicates connected by boolean operators. Moreover, the argument variables of the predicates within a given expression need not be the same, and in general cannot be the same.

If we were to define 'while' schemas in the same manner as do Ashcroft and Manna, we would defeat our purpose in restricting attention to monadic schemas in the first place, since transfer-type instructions would have to contain non-monadic predicate structures rather than simple predicates. What we have chosen to do instead is define a class of strictly monadic 'while' schemas, the <u>iteration schemas</u> of the next section, which correspond precisely to the structured schemas we have defined above. It should be emphasized, however, that unlike the 'while' schemas of Ashcroft and Manna, the iteration schemas form a strictly proper subset of the monadic program schemas.

In the following section, we formalize the notion of iteration schema. The correspondence between these schemas and the class of structured program schemas is established in Section 2.3.

2.2 Iteration Schemas.

Figure 6 illustrates an iteration schema representing a computation functionally equivalent to that represented by the structured MPS of Fig. 5.

2.2.1 Formal Definitions.

A (monadic) iteration schema is a five-tuple

$$S = (V, F, P, \Sigma, \mathcal{P})$$

where:

V is a finite set of variable symbols.

F is a finite set of function symbols.

P is a finite set of predicate symbols.

Σ is a finite set of iteration schemas, the subschemas of S.

$\mathcal{P}$ is the program of S, a finite sequence of statements of the following types:

(1) Assignment statement of the form

$$x \leftarrow f(y)$$

or of the form

$$x \leftarrow y$$

where x and y are variable symbols and f is a function symbol.

(2) Conditional statement of the form

$$\text{IF } p(x) \text{ THEN } S_t \text{ ELSE } S_f$$

where p is a predicate symbol, x is a variable symbol (the test variable of the statement), and S_t and S_f are subschemas (the true and false subschemas of the statement, respectively).

$S = (\{x,y,z\}, \{f,g\}, \{p,q,r\}, \{S_1,\ldots,S_7\}, \mathcal{P})$

$\mathcal{P}$: $x \leftarrow y$
WHILE $p(x)$ DO S_1
$y \leftarrow g(y)$
IF $p(y)$ DO S_2 ELSE S_3

$S_1 = (\{x,y,z\}, \{f,g\}, \{q,r\}, \{S_4,\ldots,S_7\}, \mathcal{P}_1)$

$\mathcal{P}_1$: WHILE $q(y)$ DO S_4
WHILE $q(x)$ DO S_5

$S_2 = (\{y,z\}, \{f,g\}, \emptyset, \emptyset, \mathcal{P}_2)$

$\mathcal{P}_2$: $y \leftarrow g(y)$
$z \leftarrow f(z)$

$S_3 = (\{y,z\}, \{f\}, \emptyset, \emptyset, \mathcal{P}_3)$

$\mathcal{P}_3$: $y \leftarrow f(y)$
$z \leftarrow f(z)$

$S_4 = (\{y,z\}, \{f,g\}, \{r\}, \{S_6,S_7\}, \mathcal{P}_4)$

$\mathcal{P}_4$: IF $r(y)$ THEN S_6 ELSE S_7

$S_5 = (\{x,y,z\}, \{f,g\}, \emptyset, \emptyset, \mathcal{P}_5)$

$\mathcal{P}_5$: $x \leftarrow g(x)$
$y \leftarrow f(y)$
$z \leftarrow f(z)$

$S_6 = (\{y,z\},\{f,g\}, \emptyset, \emptyset, \mathcal{P}_6)$

$\mathcal{P}_6$: $y \leftarrow g(y)$
$z \leftarrow f(z)$

$S_7 = (\{y,z\}, \{f\}, \emptyset, \emptyset, \mathcal{P}_7)$

$\mathcal{P}_7$: $y \leftarrow f(y)$
$z \leftarrow f(z)$

FIGURE 6. Iteration Schema Equivalent to the Schema of Figure 5.

(3) Iteration statement of the form

WHILE p(x) DO S_i

or of the form

UNTIL p(x) DO S_i

where p is a predicate symbol, x is a variable symbol (the test variable of the statement), and S_i is a subschema (the body of the statement).

If $S' = (V', F', P', \Sigma', \mathcal{P}')$ is a subschema of S, then we require that $V' \subset V$, $F' \subset F$, $P' \subset P$, and $\Sigma' \subset \Sigma$. We do not permit recursion in iteration schemas: no iteration schema may be a subschema of itself.

Interpretations are defined for iteration schemas in a manner identical to that for program schemas. Each interpretation defines an execution by the schema in a manner similar to that for program schemas:

We associate the specified domain elements with the variable symbols of the schema, and then execute sequentially the statements in the schema's program. New values are associated with variable symbols as indicated by assignment statements, and the programs of appropriate subschemas are executed in accordance with the outcomes of predicate applications made while executing conditional or iteration statements. The execution terminates when and if the execution of the final statement in the schema's program is completed. If the execution terminates, we take as the value of the schema for the execution (and corresponding interpretation) the set of elements associated with the variable symbols of the schema at the end of the execution;

the value of the schema is undefined if the execution fails to terminate.

Free interpretations are defined for iteration schemas as in Section 2.1.3.

2.2.2 Classes of Iteration Schemas.

Free iteration schemas, independent location iteration schemas, and restricted location iteration schemas are defined in manners analogous to those for program schemas.

We define two additional classes of iteration schemas which will be of interest in subsequent discussions:

An iteration schema is conditional-free if no conditional statement appears in its program, and no conditional statement appears in the program of any of its subschemas.

An iteration schema is a single appearance schema if no predicate symbol appears in more than one statement in the collection of programs belonging to the schema and its subschemas.

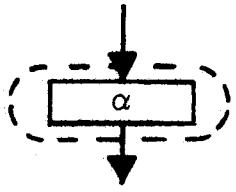
2.3 Structured Monadic Program Schemas and Iteration Schemas: A Correspondence.

We have asserted that the iteration schemas of the preceding section are an alternate formalization of the structured monadic program schemas of Section 2.1.5. In this section, we describe informally effective procedures for proceeding from one formalization to the other. Of course, we have not yet made precise the notion of schema equivalence, but the procedures are highly intuitive and are correct for virtually any reasonable notion of equivalence.

It is quite an easy task to construct from an arbitrary iteration schema a functionally equivalent structured MPS:

Let S be an arbitrary iteration schema and suppose that we are able to express each subschema of S as an equivalent composition of simple blocks. Then we can express each statement in the program of S as an equivalent composition of simple blocks, as follows:

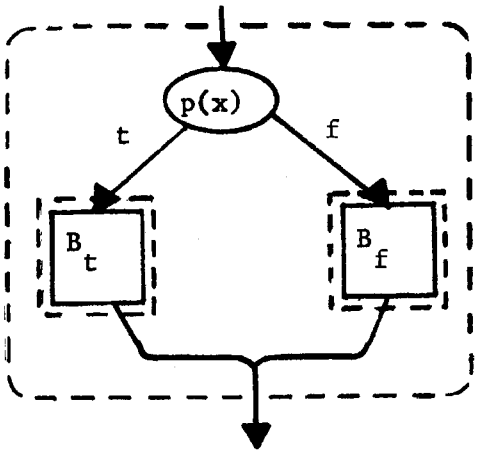
(1) If the statement is an assignment statement α, then the equivalent composition of blocks is:



(2) If the statement is a conditional statement of the form

$$\text{IF } p(x) \text{ THEN } S_t \text{ ELSE } S_f$$

then the equivalent composition of blocks is:

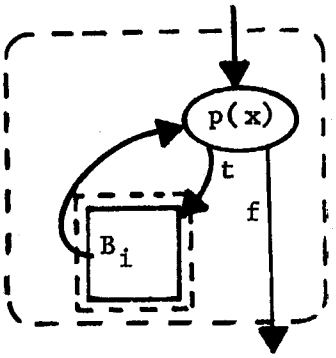


Where B_t (B_f) is the composition of simple blocks equivalent to subschema S_t (S_f).

(3) If the statement is an iteration statement of the form

WHILE p(x) DO S_i

then the equivalent composition of simple blocks is:



Where B_i is the composition of simple blocks equivalent to subschema S_i.

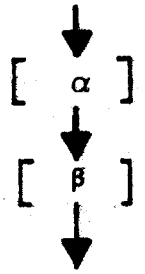
The last case (the statement is an UNTIL statement) is similar to the third.

The structured MPS equivalent to the schema S is formed quite simply by connecting sequentially the compositions of simple blocks corresponding to each of the elements in the program of S.

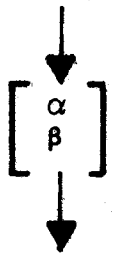
It is now clear that we can construct an equivalent structured MPS from any iteration schema — we need only work from the inside out, i.e. we need only begin with subschemas composed solely of assignment statements, then those composed of these schemas and assignment statements, etc.

The reverse construction is equally straightforward:

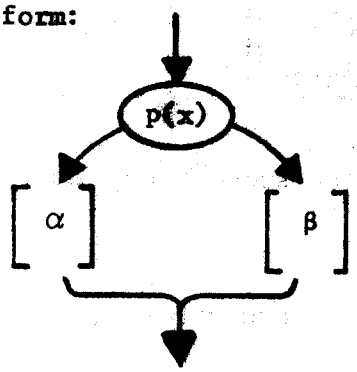
We begin by replacing each rectangle in the flowchart of a structured MPS by its contents enclosed in brackets. (The brackets will denote a portion of the flowchart which has been converted to equivalent iteration schema form.) We then merge the contents enclosed in adjacent brackets, i.e.



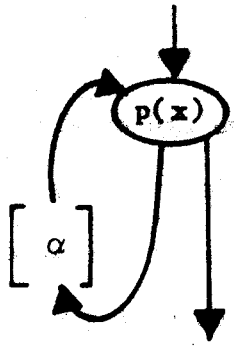
becomes



When we have completed this merging, either the entire flowchart will consist of a bracketed expression (followed by an END node), in which case we have completed the construction, or the flowchart will contain structures of the form:



or of the form:



We replace the first type of structure with:

$\lceil$IF p(x) THEN S_α ELSE S_β $\rceil$ where S_α becomes the subschema with program α, and S_β becomes the subschema with program β.

and the second with:

$\lceil$WHILE p(x) DO $S_\alpha\rceil$ if b is 't', or $\lceil$UNTIL p(x) DO $S_\alpha\rceil$ if b is 'f',

where S_α becomes the subschema with program α.

This procedure is repeated recursively until the entire remaining "flowchart" consists of a single bracketed expression followed by an END node. In such a case, the expression in brackets is the program of the iteration schema equivalent to the original structured program schema, with subschemas as given above.

2.4 Incompleteness of Iteration Schemas.

Our reason for selecting iteration schemas as the primary computational model of the thesis is that we wish to deal with equivalence issues for a class of schemas less "powerful" than the class of monadic program schemas, but still general in its ability to represent computation. Iteration schemas clearly satisfy the latter criterion, but we might question whether they satisfy the former. Clearly there exist monadic program schemas which are not structured, but do the structured schemas represent a complete subset of the monadic program schemas? In particular, we might ask if the presence of boolean operators is really essential for the simulation of arbitrary

MPS's by 'while' schemas, or whether any MPS may be simulated by an iteration schema.

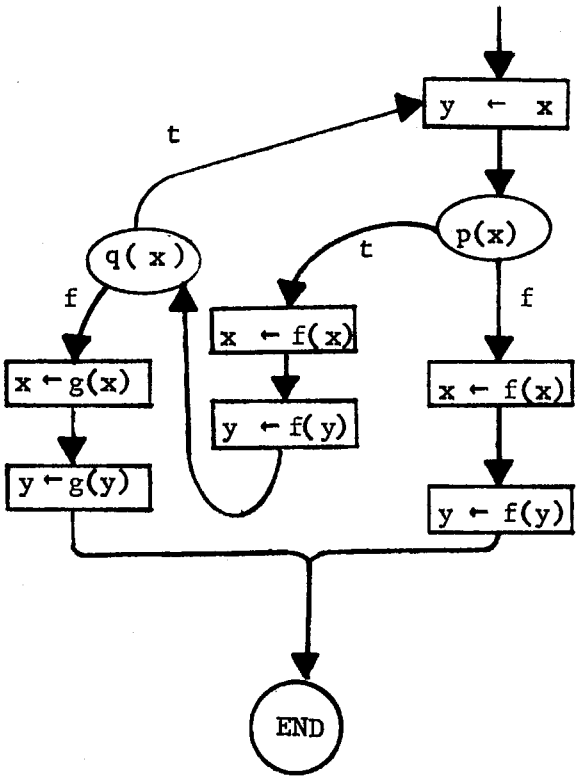
In this final section of the chapter, we demonstrate that such operators are indeed essential for the simulation of arbitrary MPS's by 'while' schemas. (In the following chapter, we define formally schematic equivalence. For the purpose of the following discussion we state that a pair of schemas are not equivalent if there exists an interpretation for the schemas such that the execution of one schema terminates, while that of the other schema fails to terminate.)

Consider the MPS of Fig. 7(a). It is intuitively obvious (and can be derived immediately from results in [14]) that if S is equivalent to some iteration schema, it is equivalent to some iteration schema which has predicate symbols {p,q} and function symbols {f,g}.

We note that an execution of S is guaranteed to diverge if it is defined by an interpretation in which the predicate associated with p and that associated with q are each identically true. Let I denote the free interpretation which satisfies this condition.

Now, suppose that there exists an iteration schema S' equivalent to S. S' must contain at least one WHILE statement s containing one of the predicate symbols, say p, such that s is executed during the execution of S' defined by I. Let D denote the domain of I, and let X denote the set of elements of D at which the predicate associated with q

(a) S:



(b)

$$\text{IF } p(x) \text{ THEN } \left[\begin{array}{l} x \leftarrow f(x) \\ y \leftarrow x \\ \text{WHILE } (p(x) \quad q(y)) \text{ DO } \left[\begin{array}{l} x \leftarrow f(x) \\ y \leftarrow x \end{array}\right] \\ \text{IF } p(x) \text{ THEN } \left[\begin{array}{l} x \leftarrow g(x) \\ y \leftarrow x \end{array}\right] \text{ ELSE } \left[\begin{array}{l} x \leftarrow f(x) \\ y \leftarrow x \end{array}\right] \end{array}\right] \text{ELSE } \left[\begin{array}{l} x \leftarrow f(x) \\ y \leftarrow x \end{array}\right]$$

FIGURE 7. Unstructured Schema and Equivalent 'While' Schema With Boolean Operators.

is evaluated prior to the first execution of statement s. Let I' be the free interpretation differing from I only in that Π_q' is associated with q, where Π_q' is the predicate defined as follows:

$$\Pi_q'(\delta) = \underline{\text{true}}, \ \delta \in X$$
$$\Pi_q'(\delta) = \underline{\text{false}}, \ \delta \notin X$$

The execution of S' defined by I' clearly cannot terminate since the behavior of S' for I and I' is identical until statement s is executed, whereupon S' must diverge because the predicate associated with p is identically true. But the execution of S will terminate as soon as Π_q' is evaluated at an element of (D-X), and since X is finite and I' is a free interpretation, Π_q' must eventually be evaluated at an element of (D-X), contradicting the equivalence of the two schemas.

A 'while' schema containing boolean operators is shown in part (b) of Fig. 7; the reader may verify that this schema is equivalent to the schema S.

CHAPTER III: EQUIVALENCE

3.1 Equivalence — Introduction.

Intuitively, we wish to consider a pair of schemas equivalent if and only if the value of each schema is the same whenever both schemas have been provided with the same interpretation. This notion of equivalence is rather strong. In particular, it is a much stronger notion than would be desirable if we intended to apply schematological results to interpreted models, since it provides no mechanism for the specification of any relations which might exist among the functions and predicates comprising the schema primitives. But this is precisely what makes it a logical notion of schematological equivalence: equivalent schemas are required to exhibit the same "black box" behavior, i.e. they are required to exhibit, for each interpretation, the same <u>output</u> behavior, with no constraints on their <u>internal</u> behavior.

The notion of equivalence as we have presented it does, however, have a serious drawback: it makes sense only if applied to pairs of schemas which have the same sets of predicate symbols, function symbols, and variable symbols. This is indeed unfortunate, since we would certainly like to consider the schemas S and S' of Fig. 8 equivalent, despite the fact that S' contains a function symbol and a predicate symbol not contained in S. Likewise, we might wish to add variable symbols to a schema in order to remember intermediate values during an execution, without considering the affects of these new symbols on the resultant schema value. (In fact, the addition of such variable symbols

$S = (\{x,y\}, \{f\}, \emptyset, \emptyset, \mathcal{P})$

where $\mathcal{P}$ is: $x \leftarrow f(x)$

$y \leftarrow f(x)$

$S' = (\{x,y\}, \{f,g\}, \{p\}, \{S_1,S_2\}, \mathcal{P}')$

where $\mathcal{P}'$ is: $x \leftarrow f(x)$

$y \leftarrow g(y)$

IF $p(y)$ THEN S_1 ELSE S_2

and $S_1 = (\{x,y\}, \{f\}, \emptyset, \emptyset, \mathcal{P}_1)$

where $\mathcal{P}_1$ is: $y \leftarrow f(x)$

and $S_2 = (\{x,y\}, \{f\}, \emptyset, \emptyset, \mathcal{P}_2)$

where $\mathcal{P}_2$ is: $y \leftarrow x$

$y \leftarrow f(y)$

FIGURE 8. Equivalent Schemas With Distinct Sets of Predicate Symbols and Function Symbols.

is necessary, in general, if one wishes to transform an arbitrary MPS into an equivalent 'while' schema in which boolean operators are permitted.)

We may handle the problem caused by schemas with distinct sets of predicate symbols and function symbols by considering extensions of interpretations. If we permit an interpretation for a schema to associate predicates and functions with symbols which do not appear in the schema, we may extend any interpretation in such a way as to provide an interpretation for some other schema. If we begin with consistent interpretations for a pair of schemas (i.e. interpretations which do not associate distinct functions with a common function symbol or distinct predicates with a common predicate symbol), then we may easily construct an interpretation which is an extension of each of the interpretations. Thus we may consider a pair of schemas to be equivalent if, whenever they are provided with consistent interpretations, the value of the schema for its interpretation is the same in each case.

The problem posed by pairs of schemas with distinct sets of variable symbols could be handled quite easily by adding a sixth component in our definition of schema — a subset of variable symbols which would be considered output symbols, symbols whose values at the end of an execution would comprise the value of the schema for that execution. We prefer to live with the shortcomings of our present notion, however, rather than introduce such a component, since the presence of non-output symbols in the schemas would add in unessential but tedious ways to the

complexity of certain proofs in subsequent chapters. Also, we would not gain much by its introduction, since most results concerning the decidability of equivalence for classes of schemas with "total output" imply the same results for corresponding classes of schemas with specified output symbols. In particular, suppose we have a pair of schemas with a common set of output symbols, plus some other variable symbols which may be different in each case. We can add additional variable symbols to each schema to ensure that the set of such symbols is the same in each case, and then add to the end of each program the statement 'x ← y', for each non-output symbol x and some specified output symbol y. Clearly, the resultant schemas will be equivalent in the sense we have proposed if and only if the original schemas are equivalent when attention is restricted to particular output symbols.

Earlier we defined the class of free interpretations for schemas, and noted that each free interpretation represents symbolically a family of closely related interpretations. It would be quite convenient if we could restrict attention to free interpretations in our discussions of equivalence, since such interpretations have several desirable properties. For one thing, all free interpretations for a schema have the same domain and interpret each function symbol in the same manner. Hence, we may specify a free interpretation solely by specifying the way it interprets predicate symbols. Also, during an execution defined by a free interpretation for a schema, identical values may be generated only if they are generated via precisely the same sequence of function applications, and the value itself makes explicit this sequence. In

addition, the application of a function to a value during such an execution must result in a different value, and this value must be longer than the original when viewed as a string of symbols. This last property is particularly significent when dealing with restricted location schemas, since it implies that the sequence of values associated with any variable symbol during such an execution will be a sequence of strings of strictly increasing length, each a proper suffix of the next. Finally, and perhaps most importantly, we can exploit the property of freeness in a schema only if we are dealing with executions of the schema under free interpretations.

Fortunately, it is easy to show that a pair of schemas are equivalent for all interpretations if and only if they are equivalent for all free interpretations. It is only necessary to show that if there exist consistent interpretations which demonstrate the non-equivalence of two schemas, then the free interpretations which represent them symbolically are also consistent and also demonstrate the non-equivalence of the schemas. The proof of the first part of the statement is trivial; the proof of the second part is by induction on the lengths of the final values of variables for the executions defined by the free interpretations, and is also straightforward. The reader is referred to [14] for details.

3.2 Strong Equivalence — Definition.

We term the notion of equivalence we have just discussed strong equivalence, since it is strong enough to imply most other reasonably defined forms of equivalence. Formally:

Let I be an interpretation for some schema, and let D be the domain of the interpretation. Let F be a set of function symbols and P a set of predicate symbols which are not interpreted by I. Then a (P,F)-extension of I is the object formed by adjoining to I, for each symbol p in P, a total predicate Π_p: D → {true,false}, and for each symbol f in F, a total function φ_f: D → D.

Let S_1 and S_2 be arbitrary schemas with possibly distinct sets of function or predicate symbols. Let F_1 and P_1 denote the sets of function symbols and predicate symbols, respectively, which appear in S_1 but not in S_2. Similarly, let F_2 and P_2 denote the sets of function symbols and predicate symbols which appear in S_2 but not in S_1. Then an interpretation I_1 for S_1 is consistent with an interpretation I_2 for S_2 if some (P_1,F_1)-extension of I_2 is a (P_2,F_2)-extension of I_1.

A schema S_1 is strongly equivalent to a schema S_2 if the value of S_1 for a free interpretation I_1 is the same as the value of S_2 for a free interpretation I_2, whenever I_1 and I_2 are consistent.

Unless otherwise noted, equivalence shall refer to strong equivalence in the remainder of this thesis.

3.3 Weak Equivalence — Definition.

Several other notions of equivalence have been proposed in the literature, one of which will be of interest later when we explore the way in which equivalence problems for structured and unstructured independent location schemas are related:

Schemas S_1 and S_2 are <u>weakly equivalent</u> if the value of S_1 for an interpretation I_1 is the same as the value of S_2 for an interpretation I_2, whenever I_1 and I_2 are consistent free interpretations and both values are defined.

It should be noted that weak equivalence is not, in fact, an equivalence relation. In particular, the relation is not transitive since any pair of schemas are each equivalent to one whose executions diverge regardless of interpretation.

CHAPTER IV: UNSOLVABLE PROBLEMS

Luckham, Park, and Paterson have demonstrated in [16] that the equivalence problem for monadic program schemas is recursively unsolvable. Briefly, thay have demonstrated that the schemas are capable of simulating two-head finite automata, and have noted that the equivalence problem for the automata was shown unsolvable in [25]. The schemas which they constructed to simulate the automata were independent location schemas except for an initial assignment instruction of the form 'x ← y' (which corresponds to the placing of both heads on the tape of an automaton) and we have, therefore, that equivalence is undecidable for restricted location monadic program schemas.

The example in Section 2.4 demonstrates that restricted location iteration schemas are strictly less powerful than restricted location MPS's. Moreover, it can be shown (cf, [13]) that there exist restricted location MPS's which cannot be simulated by restricted location 'while' schemas, even if we permit boolean expressions in conditional and iteration statements. (Informally, the reason we cannot, in general, construct such a simulating schema is that the flowchart of an arbitrary restricted location MPS may contain a directed cycle with many different exits. To place such a cycle in 'whilish' form requires merging all of these exits into one, and "remembering" the values of relevent variables at points in the new cycle which correspond to the exit points in the original cycle. When the merged exit is taken, the correct values may then be associated with each variable symbol, based on which of the exits

would actually have been taken in the original flowchart. The appropriate values are remembered by assigning them to new variable symbols, and it is precisely these assignments which prevent the resultant schema from being a restricted location schema, since they must be made within the cycle at points corresponding to the original exits.)

It is clear, therefore, that structured restricted location schemas are significently less powerful that restricted location schemas in general, and we might hope that the added structure is sufficient to render equivalence decidable for such schemas.

Unfortunately, this does not prove to be the case. In fact, we are able to prove that equivalence is undecidable even if we look at restricted location schemas which are "totally" structured — those in which even conditional statements have been removed.

In this chapter we prove that equivalence is undecidable for restricted location iteration schemas, and that it is also undecidable for such schemas which are conditional-free. For lucidity, we present the result for the more general class of schemas first, and then discuss the way in which the procedure may be strengthened to establish the result for the conditional-free schemas.

4.1 Post's Correspondence Problems.

We begin by considering two well-known unsolvable problems: the Post Correspondence Problem and the Modified Post Correspondence Problem.

A Post Correspondence Problem is an ordered pair

$$C = (A,B)$$

where $A = \{ \omega_1, \omega_2, \ldots, \omega_k \}$ $\qquad 1 \le i \le k, \quad \omega_i, \gamma_i \in \{0,1\}^*$

$B = \{ \gamma_1, \gamma_2, \ldots, \gamma_k \}$

and $\omega_i = s_{i_1} \cdot s_{i_2} \cdot \ldots \cdot s_{i_{\delta_i}}$ $\qquad s_{i_j}, r_{i_j} \in \{0,1\}$

$\gamma_i = r_{i_1} \cdot r_{i_2} \cdot \ldots \cdot r_{i_{\sigma_i}}$

We say that the Post Correspondence Problem C has a solution if and only if there exists a sequence of integers $i_1, \ldots, i_m$, $m \ge 1$, $1 \le i_n \le k$, $1 \le n \le m$, such that

$$\omega_{i_1} \cdot \omega_{i_2} \cdot \ldots \cdot \omega_{i_m} = \gamma_{i_1} \cdot \gamma_{i_2} \cdot \ldots \cdot \gamma_{i_m}$$

A Modified Post Correspondence Problem is a pair (C, ℓ), where C is a Post Correspondence Problem as above, and ℓ is an integer, $1 \le \ell \le k$. We say that the modified problem has a solution if and only if there exists a sequence of integers $i_1, \ldots, i_m$, $m \ge 0$, $1 \le i_n \le k$, $1 \le n \le m$, such that

$$\omega_\ell \cdot \omega_{i_1} \cdot \omega_{i_2} \cdot \ldots \cdot \omega_{i_m} = \gamma_\ell \cdot \gamma_{i_1} \cdot \gamma_{i_2} \cdot \ldots \cdot \gamma_{i_m}$$

It is recursively undecidable whether a given Post Correspondence Problem or a given Modified Post Correspondence Problem has a solution. (See, for example, [8].)

4.2 A Note on Notation.

We will generally define schemas in terms of their programs and the programs of their subschemas only, the function, predicate, and variable symbols being defined implicitly as those appearing in the programs. In most cases, in fact, we will not bother to distinguish between a schema and its program, referring to either as "schema".

When convenient, we will permit compound boolean expressions to appear in conditional statements of an iteration schema. We note that

IF ($\neg$b) THEN S_t ELSE S_f

for any boolean expression b is equivalent to

IF b THEN S_f ELSE S_t

and that

IF (b $\vee$ b') THEN S_t ELSE S_f

for any boolean expressions b and b' is equivalent to

IF b THEN S_t ELSE S'

where S' is

IF b' THEN S_t ELSE S_f

and we are assured, therefore, that permitting boolean expressions in conditional statements is indeed a notational convenience, and does not in any way alter the class of computations representable by iteration schemas.

4.3 Undecidability of Equivalence for Restricted Location Iteration Schemas.

Let C be a Post Correspondence Problem as defined in Section 4.1. We show how to construct a restricted location iteration schema S with the property that some free interpretation I defines a terminating execution of S if and only if the problem C has a solution.

Let S_E be the iteration schema whose program is empty, and let $S_\uparrow$ be the schema:

WHILE p(x) DO S_E

UNTIL p(x) DO S_E

for some predicate symbol p and some variable symbol x. That is, $S_\uparrow$ is a schema for which each interpretation defines a non-terminating execution.

Let p_0 and p_1 be distinct predicate symbols different from p, and let S_0 be the schema:

IF $(p_0(x) \wedge (\neg p_1(x)))$ THEN S_E ELSE $S_\uparrow$

and let S_1 be the schema:

IF $(p_1(x) \wedge (\neg p_0(x)))$ THEN S_E ELSE $S_\uparrow$

Suppose we interpret the predicate symbols p, p_0, and p_1 over some domain D. We may consider an element of the domain as representing the symbol '0' if, when this element is initially associated with symbol x, the execution of S_0 defined by the interpretation terminates. Similarly, we may consider the element as representing the symbol '1' if, when the element is initially associated with x, the execution of S_1 defined by

the interpretation terminates. Viewed in this way, no element of D may represent both '0' and '1', although some elements may represent neither symbol.

For each i, $1 \leq i \leq k$, let A_{ω_i} be the schema:

IF q(x) THEN $S_{s_{i_1}}$ ELSE $S_{\uparrow}$
x ← f(x)
IF q(x) THEN $S_{s_{i_2}}$ ELSE $S_{\uparrow}$
x ← f(x)
.
.
.
IF q(x) THEN $S_{s_{i_{\delta_i}}}$ ELSE $S_{\uparrow}$
x ← f(x)

(In the above, q is some new predicate symbol and $S_{s_{i_j}}$ denotes, for each j, the schema S_0 or the schema S_1, according as s_{i_j} is 0 or 1.)

In this manner we associate a schema with each string of symbols in the first component of the problem C. If we provide an interpretation with some domain D for the schema A_{ω_i}, then the execution of A_{ω_i} defined by the interpretation will diverge unless the sequence of symbols represented (in the way described above) by the elements ν, $\varphi_f(\nu)$, $\varphi_f(\varphi_f(\nu))$, ... , $\varphi_f^{\delta_i - 1}(\nu)$ comprise the string ω_i, where $\nu \in D$ is the value initially associated with x and φ_f: D → D is the function associated with f by the interpretation. The execution will also diverge if any evaluation of the predicate associated with q has outcome <u>false</u>

during the execution. (The reason for the inclusion of q in the schema will become apparant later.)

Let S_0' be the schema identical to S_0 except with variable symbol y in place of variable symbol x; let S_1' be the schema identical to S_1 except with y in place of x.

For each i, $1 \leq i \leq k$, let B_{γ_i} be the schema:

IF q(y) THEN $S_{r_{i_1}}'$ ELSE $S_\uparrow$
$y \leftarrow f(y)$
IF q(y) THEN $S_{r_{i_2}}'$ ELSE $S_\uparrow$
$y \leftarrow f(y)$
.
.
.
IF q(y) THEN $S_{r_{i_{\sigma_i}}}'$ ELSE $S_\uparrow$
$y \leftarrow f(y)$

(In the above, $S_{r_{i_j}}'$ denotes, for each j, the schema S_0' or the schema S_1', according as r_{i_j} is 0 or 1.)

These schemas correspond to the strings in the second component of C in the same way as those defined previously correspond to the strings in the first.

For each i, $1 < i < k$, let R_i be the schema:

IF $t_i(z)$ THEN A_{ω_i} ELSE S_E
IF $t_i(z)$ THEN B_{γ_i} ELSE R_{i+1}

where each t_i is a distinct new predicate symbol.

Let R_1 be the schema:

IF $t_1(z)$ THEN $A\omega_1$ ELSE S_E

IF $t_1(z)$ THEN $B\gamma_1$ ELSE R_2

$z \leftarrow f(z)$

and let R_k be the schema:

IF $t_k(z)$ THEN $A\omega_k$ ELSE S_E

IF $t_k(z)$ THEN $B\gamma_k$ ELSE $S_\uparrow$

Finally, let S be the schema:

$y \leftarrow x$

IF $p_I(z)$ THEN R_1 ELSE R_1

WHILE $p_I(z)$ DO R_1

IF $q(x)$ THEN $S_\uparrow$ ELSE S_E

IF $q(y)$ THEN $S_\uparrow$ ELSE S_E

We claim that some execution of S under a free interpretation will terminate if and only if the Post Correspondence Problem C has a solution. In particular, suppose that some such execution of S terminates. Let n denote the number of times during the execution of S that the subschema R_1 is executed, and let $t_{j_1}, t_{j_2}, \ldots, t_{j_n}$ be the sequence of t's whose applications resulted in <u>true</u> outcomes during the executions of R_1. It is clear from the construction that the value of x at the end of the execution of S must be $f^{m_1} \cdot \Delta_x$, where $m_1 = |\omega_{j_1} \cdot \omega_{j_2} \cdot \ldots \cdot \omega_{j_n}|$, and that the value of y at the conclusion of

the execution must be $f^{m_2}\cdot\Delta_x$, where $m_2 = |\gamma_{j_1}\cdot\gamma_{j_2}\cdot\ldots\cdot\gamma_{j_n}|$. Also, the sequence of symbols represented by the elements Δ_x, $f\cdot\Delta_x$, ... , $f^{m_1}\cdot\Delta_x$ must comprise $\omega = \omega_{j_1}\cdot\omega_{j_2}\cdot\ldots\cdot\omega_{j_n}$, otherwise the execution of S would have diverged at the execution of some subschema A_{ω_i} when it was discovered that the wrong symbol, or no symbol at all, was represented by an element. Similarly, the sequence of symbols represented by Δ_x, $f\cdot\Delta_x$, ... , $f^{m_2}\cdot\Delta_x$ must comprise $\gamma = \gamma_{j_1}\cdot\gamma_{j_2}\cdot\ldots\cdot\gamma_{j_n}$, and therefore it must be the case that ω is a prefix of γ, or vice versa.

But it must also be the case that m_1 and m_2 are identical:

The value associated with x is tested with the predicate corresponding to symbol q each time a statement of the form 'x ← f(x)' is executed, and similarly for the values associated with y. In each case but the last the outcome of the test is <u>true</u>, otherwise the execution would have diverged. Now, suppose that m_1 and m_2 are not the same; in particular, suppose that $m_1 > m_2$. It must be the case that $\Pi_q(f^{m_2}\cdot\Delta_x)$ is <u>true</u> (where Π_q is the predicate associated with q). But then the execution of S would have diverged when the fifth statement in its program was executed, a contradiction. Hence $m_1 = m_2$.

But then it must be the case that $\omega = \gamma$, and thus j_1, ... , j_n must be a solution to the correspondence problem C.

The remaining argument, that some execution of S under a free interpretation terminates if C has a solution, is also quite easy. Suppose that $i_1, \ldots, i_m$ is a solution for C. Then we provide S with the free interpretation that associates with p_0 the predicate:

$$\Pi_{p_0}(f^n \cdot \Delta_x) = \underline{\text{true}} \text{ if the } (n+1)\text{st symbol of } \omega_{i_1} \cdot \ldots \cdot \omega_{i_m} \text{ is 0, or } \underline{\text{false}} \text{ otherwise}$$

that associates with p_1 the predicate:

$$\Pi_{p_1}(f^n \cdot \Delta_x) = \underline{\text{true}} \text{ if the } (n+1)\text{st symbol of } \omega_{i_1} \cdot \ldots \cdot \omega_{i_m} \text{ is 1, or } \underline{\text{false}} \text{ otherwise}$$

that associates with q the predicate:

$$\Pi_q(f^n \cdot \Delta_x) = \underline{\text{false}} \text{ if } n = |\, \omega_{i_1} \cdot \ldots \cdot \omega_{i_m} |, \text{ or } \underline{\text{true}} \text{ otherwise}$$

that associates with t_j, $1 \le j \le k$, the predicate:

$$\Pi_{t_j}(f^n \cdot \Delta_z) = \underline{\text{true}} \text{ if } j = i_{n+1},\ 1 \le n \le m, \text{ or } \underline{\text{false}} \text{ otherwise}$$

that associates with p_I the predicate:

$$\Pi_{p_I}(f^n \cdot \Delta_z) = \underline{\text{true}} \text{ if } n < m, \text{ or } \underline{\text{false}} \text{ otherwise}$$

and that associates with p an arbitrary predicate.

The reader will verify that the execution of S defined by the above interpretation is guaranteed to terminate.

From the above construction we have:

Theorem 4.1:

The equivalence problem for restricted location iteration schemas is unsolvable.

Proof: Suppose that the equivalence problem were solvable. Given an arbitrary Post Correspondence Problem C we could construct a restricted location iteration schema S as outlined above. The correspondence problem would have a solution if and only if S were not equivalent to the schema $S_{\uparrow}$, and we would therefore be able to decide if C has a solution, contradicting the known unsolvability of Post's Correspondence Problem.

□

4.4 Undecidability of Equivalence for Conditional-Free Schemas.

In this section we describe how a conditional-free, restricted location schema may be constructed from an arbitrary Modified Post Correspondence Problem in such a way that some execution of the schema terminates if and only if the correspondence problem has a solution.

Let C be an arbitrary Post Correspondence Problem as defined in Section 4.2, and let $C' = (C, \ell)$ be a modified correspondence problem, where ℓ is an integer between 1 and k, inclusive. We begin by modifying some of the subschemas of the preceding section:

We note first that the schema S_0 is equivalent to the conditional-free schema:

UNTIL $p_0(x)$ DO S_E

WHILE $p_1(x)$ DO S_E

Similarly, the schema S_1 is equivalent to:

UNTIL $p_1(x)$ DO S_E

WHILE $p_0(x)$ DO S_E

For each i, the subschema A_{ω_i} consists of assignment statements and conditional statements of the form:

IF $q(x)$ THEN $S_{s_{i_j}}$ ELSE $S_\uparrow$

Clearly, we may replace each such conditional statement with the statement:

UNTIL $q(x)$ DO S_E

followed by the sequence of statements which makes up the program of the

subschema $S_{s_{i_j}}$. Hence, each such schema A_{ω_i} can be transformed into a conditional-free schema. In a similar manner, each subschema of the form B_{γ_i} can be transformed into a conditional-free schema.

For each i, $1 \leq i \leq k$, let D_i be the schema consisting of the statements of the schema A_{ω_i}, followed by the statements of the schema B_{γ_i}, followed by the statement:

$$z \leftarrow f(z)$$

Each D_i can be made conditional-free by using the conditional-free schemas equivalent to A_{ω_i} and B_{γ_i}.

Let R be the schema

WHILE $t_1(z)$ DO D_1

WHILE $t_2(z)$ DO D_2

.
.
.

WHILE $t_k(z)$ DO D_k

Finally, let S' be the schema:

$y \leftarrow x$

α

β

WHILE $p_I(z)$ DO R

WHILE $q(x)$ DO S_E

WHILE $q(y)$ DO S_E

where α is the sequence of statements which makes up the schema A_{ω_ℓ},

and β is the sequence of statements which makes up the schema B_{β_ℓ}. Since each of the component schemas of S' is equivalent to a conditional-free schema, S' may certainly be made conditional-free. The reader may verify that some execution of S' defined by a free interpretation for the schema terminates if and only if the problem C' has a solution. (The argument is virtually identical to that given in the previous section.)

Thus we have:

Theorem 4.2:
The equivalence problem for restricted location, conditional-free schemas is unsolvable.

Proof: The decidability of equivalence for such schemas implies the solvability of the Modified Post's Correspondence Problem.

□

4.5 Discussion.

The schemas of Theorem 4.2 are a rather restricted class of monadic program schemas, more restricted than any class for which the equivalence problem has previously been shown unsolvable. This is not actually surprising, however, since the effort which has been expended in the study of equivalence problems for schematized models has been directed primarily towards finding broad classes of such models for which equivalence is decidable, rather than restricted classes for which it is

not. The paucity of results which have been obtained to date, however, suggests that perhaps both directions should be explored, if only to gain some insight into the types of restrictions which are apt to lead to solvable problems.

In a modest way Theorem 4.2 provides us with some such insight, since it demonstrates that structure, or the lack thereof, is not in itself a key to equivalence problems in monadic program schemas. We are led to inquire, therefore, whether structure, taken in conjunction with other reasonable restrictions, might lead to solvable problems, or alternatively whether the constructions of the previous sections can be strengthened to yield results analogous to that of Theorem 4.2 for yet more restricted classes of schemas.

For example, what is the effect of restricting the number of distinct symbols in the schemas? The schemas in the construction of Section 4.4 employ only a single function symbol, but arbitrarily many predicate symbols. At the cost of some lucidity, however, we could present a construction involving schemas with a single function symbol and a single predicate symbol (see Appendix A), and thus restricting the number of predicate and function symbols helps us little in our quest for solvable problems, even if dealing with totally structured schemas. Matters are slightly different with respect to variable symbols, however: while it is an easy matter to present a construction employing only two variable symbols (we merely use x or y in place of z), no further reduction is possible in the construction, since the

equivalence problem for schemas with a single variable symbol reduces to that for finite state automata.

Prompted in part by interesting results for certain interpreted models for computation [17, 28], we might ask about the effect of limiting the number of levels of iteration in structured schemas. We note, for instance, that the construction of Section 4.3 employs schemas with two levels of iteration, while that of Section 4.4 employs schemas with three levels. Is equivalence decidable for structured schemas with one level of iteration? Is equivalence decidable for conditional-free schemas with two levels of iteration? (Both of these questions are open. It is known, however, that equivalence is decidable for conditional-free schemas with a single level of iteration, since these schemas are a subclass of the "unnested loop" schemas of Paterson [18].)

Other restrictions we consider follow from the observations that the schemas in the previous sections are not free; that they are properly restricted location, as opposed to independent location; and that certain predicate letters appear in several statements in the schemas. We might ask, therefore:

Is equivalence decidable for such schemas which are free? Which are independent location schemas? Which are single appearance schemas? Which possess various combinations of these properties?

Answers to some of these questions are presented in the following chapter.

CHAPTER V: SOLVABLE PROBLEMS

In this chapter we provide affirmative answers to some of the questions raised at the end of Chapter IV. The techniques we employ are hardly generalizable to non-structured schemas, but we are confident that they are applicable to classes of structured schemas more general than those we consider in this chapter. We are concerned, therefore, with illustrating these techniques as well as with obtaining specific results. (The first case we consider, for example, is perhaps rather more pedantic than theoretically interesting.)

Briefly, the approach is as follows:

We demonstrate that for certain classes of schemas it is possible to find iteration statements in pairs of equivalent schemas which play similar roles during consistent schema executions, i.e. statements which execute the same number of times and test the same values during such executions, and which affect in a similar manner the variable symbols in each schema. Given an arbitrary pair of schemas in the class, we identify statements which must be so related if the schemas are equivalent, and use their assumed similarity to reduce the equivalence of the schemas to that of simpler schemas, a procedure which is applied recursively until the problem has been reduced to that for schemas whose equivalence is trivially decidable.

The approach is illustrated in Section 5.5, after formalizing the notion of "similar roles" and presenting some preliminary results.

5.1 Test Sequences and Logic Equivalence.

Let S be an iteration schema with predicate symbols P, and let D be the domain of the free interpretations for S. Then the set of tests for S is the set D × P.

Suppose that s is a conditional or iteration statement in S and that E is a free execution of S (that is, an execution defined by a free interpretation for S). Then a test $\tau = (\delta, p)$ is said to be made at s during E if the predicate assigned to symbol p by the interpretation defining E is evaluated at the expression δ during the execution of statement s. (The application must be made during the execution of the statement itself, rather than during the execution of the subschema appearing in the statement.) We denote by TESTS(s,E) the sequence of tests made at s during E, and by TESTS*(S,E) the sequence of all tests made during E (including those made during the execution of statements in subschemas of S).

Let E and E' be executions of iteration schemas S and S', respectively. We say that E and E' are consistent if they are defined by consistent interpretations for the schemas. Let s be a conditional or iteration statement in S, and let s' be a conditional or iteration statement in S'. Then s and s' are logic equivalent if TESTS(s,E) = TESTS(s',E') whenever E and E' are finite, consistent executions.

Later in this chapter we show that for certain classes of schemas logic equivalent statements must appear in equivalent members of the classes. In fact, the existence of such statements forms the basis of our proofs of the decidability of equivalence for these classes.

5.2 Productivity and Essentiality.

It would be naive to assume that equivalent schemas need make precisely the same tests during consistent executions, since some of the tests made during the executions may be non-productive tests — that is, tests whose outcomes do not affect the resultant values of the schemas. For example, the value of the schema in Fig.9 for any execution is independent of the outcome of the test made at the statement labelled s. Since s is the only statement in which symbol q appears, it is clear that no test of the form (δ,q) made during an execution of S can be productive. Hence, the value of S will be the same for any pair of interpretations for the schema which differ in the way they interpret q, and we say, therefore, that q is an inessential symbol of S.

As might be expected, it is undecidable in general whether a given predicate symbol is essential in an iteration schema. In particular, if it were decidable we could easily decide whether or not arbitrary iteration schemas S_1 and S_2 were equivalent: they would be equivalent if and only if p were inessential in the schema

$$\text{IF } p(x) \text{ THEN } S_1 \text{ ELSE } S_2$$

where p is a predicate symbol not appearing in S_1 or S_2, and x is an arbitrary variable symbol.

Fortunately, the fact that essentiality is not decidable does not preclude us from using the notion to advantage, since essentiality is assured for symbols of particular interest to us later in the chapter.

$S = (\{x,y\}, \{f,g\}, \{q\}, \{S_1,S_2\}, \mathcal{P})$

where $\mathcal{P}$ is:

$x \leftarrow y$

s: IF q(x) THEN S_1 ELSE S_2

$y \leftarrow f(y)$

$S_1 = (\{x,y\}, \{f,g\}, \emptyset, \emptyset, \mathcal{P}_1)$

where $\mathcal{P}_1$ is:

$x \leftarrow f(x)$

$x \leftarrow g(x)$

$y \leftarrow f(x)$

$S_2 = (\{x,y\}, \{f,g\}, \emptyset, \emptyset, \mathcal{P}_2)$

where $\mathcal{P}_2$ is:

$y \leftarrow f(y)$

$x \leftarrow y$

$x \leftarrow g(x)$

$y \leftarrow x$

$y \leftarrow f(y)$

FIGURE 9. Schema With Inessential Predicate Symbol.

5.3 Additional Definitions and Terminology.

5.3.1 Definitions.

Let S be an iteration schema and let k be the maximum number of distinct iteration statements executed during any execution of S. Then we say that S is a schema of <u>size</u> k. (While this notion of size is certainly not the most intuitive, it shall prove quite useful in subsequent discussions.)

Let τ be a test for an iteration schema S, and let E be a free execution of S. Then τ is <u>free</u> with respect to E if $\tau \notin$ TESTS*(S,E). (Intuitively, a test is free with respect to an execution if the outcome of the test is not fixed by the execution.)

Let S be an iteration schema. Then the set of <u>main</u> statements of S is defined recursively as follows:

(i) If s is a statement in the program of S, then s is a <u>main</u> statement of S.

(ii) If s is a main conditional statement of S and s' is a statement in the program of the true or false subschema of s, then s' is a main statement of S.

Let s be a conditional or iteration statement in a schema S. A variable is said to be <u>modified</u> at s if, for some execution of the schema, the value associated with its symbol immediately prior to an execution of s is different from that associated with the symbol immediately following the execution of s. A variable is said to be <u>active</u> in s if it is the test variable of s, or it is the test variable of some statement in the

subschema(s) appearing in s; otherwise, the variable is said to be passive in s.

5.3.2 A Note on Terminology.

We shall often want to modify schema interpretations in such a way as to obtain executions differing in some respect from those defined by the original interpretations. For convenience, we shall generally express such modifications in terms of the executions themselves, rather than in terms of the defining interpretations. In particular, we shall often refer to obtaining a new execution from some given execution by changing the outcomes of particular tests, whereas what is actually meant is that we may obtain the new execution by changing appropriately the interpretation defining the original.

5.4 The Elimination Theorem.

In [24], Rodriguez studied the equivalence problem for program graphs, a schematized model for parallel computation, and proposed that one might decide the equivalence of a pair of such graphs by examining their behavior for consistent base executions, executions in which no cycle of either graph is executed more than once. Unfortunately, the proposed approach will not work if applied to our models: it is an easy task to construct a pair of iteration schemas which are not equivalent, but which have the same value for all consistent executions in which the body of no iteration statement is executed more than once.

For example, let S be the schema

$x \leftarrow f(x)$

WHILE $p(x)$ DO S_1

$x \leftarrow f(x)$

$x \leftarrow f(x)$

WHILE $p(x)$ DO S_1

where S_1 is the schema '$x \leftarrow f(x)$', and let S' be the schema

$x \leftarrow f(x)$

$x \leftarrow f(x)$

UNTIL $p(x)$ DO S_1

$x \leftarrow f(x)$

WHILE $p(x)$ DO S_1

The reader may verify that S and S' have the same value for all pairs of consistent executions in which no instance of S_1 is executed more than once. But S and S' are clearly not equivalent, since if I is the free interpretation which assigns to p the predicate Π_p defined as follows:

$$\Pi_p(f^i \cdot \Delta_x) = \underline{\text{false}},\ 0 \leq i \leq 4$$
$$\Pi_p(f^i \cdot \Delta_x) = \underline{\text{true}},\ i > 4$$

then the execution of S defined by I terminates, while that of S' must certainly diverge.

However, it <u>is</u> possible to show that if a pair of iteration schemas satisfy certain conditions, then they must contain a pair of iteration statements such that the schemas are equivalent if and only if they have the same values for all consistent executions in which the bodies of <u>these</u> statements are executed no more than once. This implies that we can decide the equivalence of such schemas by deciding the equivalence of two pairs of schemas which are "simpler" than the originals in the sense that they contain fewer iteration statements:

(i) the schemas obtained from the originals by eliminating these particular iteration statements from their respective programs; and

(ii) the schemas obtained from the originals by replacing these iteration statements with their bodies.

Moreover, if we can ensure that the resultant pairs of schemas satisfy the same conditions as the original pair, we can repeatedly

eliminate iteration statements from successive pairs of schemas until we have reduced the equivalence of the original schemas to the equivalence of some bounded number of pairs of schemas which contain no iteration statements (and for which equivalence is trivially decidable). Thus the conditions for which such iteration statements are guaranteed to exist are of considerable interest.

The conditions are explicated in Theorem 5.1 in this section, after a preliminary result:

Lemma 5.1:

Let X, Y and Z be sets of words over some alphabet T. Let f_1: $X \to T^*$ and f_3: $X \to T^*$ be total functions, and let f_2: $Y^* \to T^*$ be a total function such that for each $\delta,\sigma \in Y^*$ we have $f_2(\delta)\cdot f_2(\sigma) = f_2(\delta\cdot\sigma)$. Let $\alpha \in X$ and let $\gamma \in Z$, and suppose the following equalities hold:

1. $\gamma \cdot \alpha = f_3(\alpha)\cdot f_1(\alpha)$
2. $\gamma\cdot\beta\cdot\alpha = f_3(\alpha)\cdot f_2(\beta)\cdot f_1(\alpha)$, all $\beta \in Y$

Then $\gamma\cdot\beta\cdot\alpha = f_3(\alpha)\cdot f_2(\beta)\cdot f_1(\alpha)$, all $\beta \in Y^*$.

Proof: We know from (1) and (2) that the assertion is correct if β is the empty string or β is an element of Y. Assume the assertion is correct for all $\beta \in Y^i$, $0 \le i < j$. Then the assertion is correct for all $\beta \in Y^j$, as follows:

Let $\beta = \beta_1\cdot\beta_2$, $\beta_1 \in Y^{j-1}$, $\beta_2 \in Y$. There are three cases to consider, based on the relationship of α to $f_1(\alpha)$ in equality (1):

Case 1. $\alpha = f_1(\alpha)$

Then $\gamma = f_3(\alpha)$. We have from equality (2):

$\gamma \cdot \beta_1 \cdot \alpha = \gamma \cdot f_2(\beta_1) \cdot \alpha$ and $\gamma \cdot \beta_2 \cdot \alpha = \gamma \cdot f_2(\beta_2) \cdot \alpha$, whence

$\beta_1 \cdot \beta_2 = f_2(\beta_1) \cdot f_2(\beta_2) = f_2(\beta_1 \cdot \beta_2)$ and thus

$\gamma \cdot \beta \cdot \alpha = \gamma \cdot f_2(\beta) \cdot \alpha$

Case 2. $\alpha = \mu \cdot f_1(\alpha)$, μ non-empty.

Then $\gamma \cdot \mu = f_3(\alpha)$. We have:

$$\forall \rho [(\gamma \cdot \rho \cdot \alpha = f_3(\alpha) \cdot f_2(\rho) \cdot f_1(\alpha)) \Rightarrow$$
$$(\gamma \cdot \rho \cdot \mu \cdot f_1(\alpha) = f_3(\alpha) \cdot f_2(\rho) \cdot f_1(\alpha)) \Rightarrow$$
$$(\gamma \cdot \rho \cdot \mu = f_3(\alpha) \cdot f_2(\rho)) \Rightarrow$$
$$(\gamma \cdot \rho \cdot \mu = \gamma \cdot \mu \cdot f_2(\rho)) \Rightarrow$$
$$(\rho \cdot \mu = \mu \cdot f_2(\rho))]$$

Substituting β_1 and β_2 for ρ yields:

$$\beta_1 \cdot \mu = \mu \cdot f_2(\beta_1)$$
$$\beta_2 \cdot \mu = \mu \cdot f_2(\beta_2)$$

Then:

$$\begin{aligned}
\gamma \cdot \beta \cdot \alpha &= \gamma \cdot \beta \cdot \mu \cdot f_1(\alpha) \\
&= \gamma \cdot \beta_1 \cdot \beta_2 \cdot \mu \cdot f_1(\alpha) \\
&= \gamma \cdot \beta_1 \cdot \mu \cdot f_2(\beta_2) \cdot f_1(\alpha) \\
&= \gamma \cdot \mu \cdot f_2(\beta_1) \cdot f_2(\beta_2) \cdot f_1(\alpha) \\
&= f_3(\alpha) \cdot f_2(\beta_1 \cdot \beta_2) \cdot f_1(\alpha) \\
&= f_3(\alpha) \cdot f_2(\beta) \cdot f_1(\alpha)
\end{aligned}$$

Case 3. $\mu\cdot\alpha = f_1(\alpha)$, μ non-empty.

The proof for Case 3 is similar to that for Case 2 and is left to the reader.

□

Theorem 5.1: (Elimination Theorem)

Let S and S' be iteration schemas with the same variable symbols, and let D be the union of the domains of their free interpretations. Let D' be the set $\{\delta \mid \delta\cdot\Delta_x \in D$, x any variable symbol of the schemas$\}$. Let s and s' be iteration statements in the programs of S and S', respectively, and suppose that for any variable symbol y of the schemas there exist total functions $f_{y_1}: D \to D$, $f_{y_2}: D' \to D'$, and $f_{y_3}: D \to D'$, which satisfy the following condition:

Let E and E' be finite, consistent, free executions of S and S'. Let α, β, and γ be strings such that α is the value associated with y just prior to the first execution of statement s during E, $\beta\cdot\alpha$ is the value associated with y immediately following the last execution of s, and $\gamma\cdot\beta\cdot\alpha$ is the value associated with y at the conclusion of E. Let α', β', and γ' be strings defined in a similar manner with E' in place of E and s' in place of s. Then:

1. $\alpha' = f_{y_1}(\alpha)$
2. $\beta' = f_{y_2}(\beta)$
3. $\gamma' = f_{y_3}(\alpha)$

Suppose also that concatenation distributes over the function f_{y_2}, i.e. suppose that $f_{y_2}(\delta)\cdot f_{y_2}(\sigma) = f_{y_2}(\delta\cdot\sigma)$, all $\delta,\sigma \in D'$.

Let S_0 be the schema obtained from S by eliminating statement s from its program. Let S_1 be the schema obtained from S by replacing s with the program of the body of s. Let S'_0 and S'_1 be the schemas obtained by eliminating s' in a similar manner from the schema S'.

Then S is equivalent to S' if and only if S_0 is equivalent to S'_0, and S_1 is equivalent to S'_1.

Proof: From the previous lemma, it must be the case that S and S' are equivalent if they have the same values for consistent executions during which the bodies of s and s' are executed no more than once. (This ensures that the two equalities of the lemma are satisfied.) Clearly this is the case if and only if S_0 is equivalent to S'_0, and S_1 is equivalent to S'_1.

□

Perhaps a few words about the theorem are in order, particularly with regard to its applications:

Suppose we have a pair of schemas S and S' whose programs consist of assignment and iteration statements only, and which have identical sets of variable symbols. Let s be the last iteration statement in the program of S, and suppose there exists a statement s' in the program of S' such that s and s' are logic equivalent. For any variable symbol x of S, we may divide the value ξ_x associated with x at the conclusion of a finite, free execution E of S into three parts: the part generated prior to the executions of statement s, the part generated during the executions of statement s, and the part generated following the executions

of statement s. (We note that the last part is a constant: since s is the last iteration statement in the program of S, the last part is determined by the particular assignment statements following s and is independent of the execution we consider.)

Similarly, we may divide the value ξ'_x associated with x at the conclusion of a consistent execution of S' into portions generated prior to the executions of statement s', during the executions of statement s', and following the executions of statement s'.

The conditions required by the theorem for the elimination of s and s' are these:

(i) The portion of ξ'_x generated during the executions of s' depends only on the portion of ξ_x generated during the executions of s, while the other portions of ξ'_x do <u>not</u> depend on this portion of ξ_x.

(ii) The portion of ξ'_x generated during <u>each</u> execution of s' depends only on the portion of ξ_x generated during the corresponding execution of statement s. (Intuitively, this ensures that concatenation distributes over the function f_2.)

These requirements are of course rather strict, but we shall show that for several interesting classes of schemas, the requirements <u>must</u> be satisfied by any pair of equivalent schemas.

5.5 Solvable Problems for Single Appearance Schemas.

In this section we demonstrate that equivalence is decidable for certain classes of restricted location, single appearance schemas (RLSA schemas). We begin by considering the class of such schemas which are conditional-free.

5.5.1 Decidability of Equivalence for Conditional-free RLSA Schemas.

Let S be an iteration schema, and let s be an iteration statement in S. We say that s is a trap in S if, whenever the first element of TESTS(s,E) has outcome true if s is a WHILE statement or outcome false if s is an UNTIL statement, E is a non-terminating execution of S. (That is, a statement in a schema is a trap if the execution of its body is sufficient to guarantee the divergence of the schema's execution.)

While it is in general undecidable whether an iteration statement in an arbitrary iteration schema is a trap, it is decidable whether an iteration statement in a RLSA schema is a trap:

Let S be such a schema. Clearly, S contains a trap if and only if it contains some iteration statement s such that the body of s contains no assignment statement of the form '$x \leftarrow f(x)$', where x is the test variable of s. If such a statement exists, then it is a trap as is any iteration statement in its body. After being identified, any such trap can be removed from the schema and the schema can be re-examined for additional traps until all have been found.

Of course we cannot, in general, remove traps from a schema without

affecting its output behavior under certain interpretations. If the schema is a RLSA schema, however, we can at least ensure that its traps take a particularly simple form:

An iteration schema is in proper form if the body of each of its traps is the empty schema S_E.

We note that any RLSA schema can be effectively transformed into an equivalent RLSA schema in proper form.

Lemma 5.2:

Let S and S' be equivalent RLSA schemas in proper form, such that neither schema contains main conditional statements. Let s be a main WHILE (UNTIL) statement in S, and let p be the predicate symbol in s. Then p appears in a main WHILE (UNTIL) statement in S'.

Proof: Suppose otherwise. In particular, suppose that p appears in a main WHILE statement in S but does not appear in a main WHILE statement in S'. Let E' be the execution of S' in which the body of no iteration statement is executed, and let E be any consistent execution of S defined by an interpretation in which the predicate assigned to p is identically true. (Since no test of the form (δ,p) is made with outcome false during E', some such E surely exists.) Then E' is a terminating execution, while E is clearly non-terminating, contradicting the equivalence of S and S'.

□

Lemma 5.3:

Let S and S' be equivalent RLSA schemas such that neither schema contains main conditional statements. Then if p is a predicate symbol in a main trap of s, p is a predicate symbol in a main trap of S'.

Proof: Since p appears in a main iteration statement in S, it must also appear (according to the previous lemma) in a main iteration statement in S'; moreover, each of these statements must be WHILE statements or each must be UNTIL statements. Let s and s' denote the statements in S and S', respectively, in which p appears.

Let E be any finite execution of S, and let E' be a consistent execution of S'. Since E is finite, TESTS(s,E) must consist of a single element τ. Clearly, τ must be in TESTS(s',E'), since we could otherwise change the outcome of τ in E to obtain a non-terminating execution of S consistent with E', contradicting the equivalence of S and S'. Also, τ must be the first element of TESTS(s',E'), since if it were not we could change the outcome of this first element to obtain a finite execution E'' of S' such that E'' is consistent with E and τ is free with respect to E''. We could then change the outcome of τ in E to obtain a non-terminating execution of S consistent with E'', again contradicting the equivalence of S and S'. Hence, for any pair of consistent finite executions by the schemas, the first test made at s must be the first test made at s'.

But then s' must be a trap:

If not, we could surely find a finite execution E'_o of S' in which the body of s' is executed at least once. Since s and s' are both WHILE statements or both UNTIL statements, the body of s must also be executed at least once during any execution E_o of S consistent with E'_o (since the first test made at s during E_o must be the same as that made at s' during E'_o). But since s is a trap, the execution E_o must be non-terminating, and we again contradict the equivalence of S and S'.

□

Lemma 5.4:

Let S and S' be equivalent RLSA schemas such that neither contains main conditional statements. Let s be a main iteration statement of S. Then there exists a main iteration statement s' of S' such that s' and s are logic equivalent.

Proof: If s is a trap, the result follows from the preceding lemma. Suppose that s is not a trap:

Let p be the predicate symbol in s. According to Lemma 5.2, S' must contain a main iteration statement s' with predicate symbol p. If s and s' are not logic equivalent, there must exist finite, consistent, free executions E of S and E' of S' such that TESTS(s,E) $\neq$ TESTS(s',E'), and we can surely find such sequences for which all but the last elements of the sequences are the same. Let τ be the last element of TESTS(s,E) and let τ' be

the last element of TESTS(s',E'). The first component of one of the tests, say τ, is at least as long as the first component of the other. We need only change the outcome of τ to <u>true</u> if s is a WHILE statement, or to <u>false</u> if s is an UNTIL statement, and provide the same outcome for each subsequent test made at s to obtain a non-terminating execution of S. But since all of these tests must be free with respect to E', the execution is consistent with E', contradicting the equivalence of S and S'.

□

<u>Lemma 5.5</u>:

Let S and S' be equivalent, conditional-free, RLSA schemas in proper form. Let s be any main iteration statement of S, and let R be the body of s. Let s' be the iteration statement in S' logic equivalent to s, and let R' be the body of s'. Then every predicate symbol in a main iteration statement of R is in some main iteration statement of R'.

<u>Proof</u>: Let S_1 be the schema obtained from S by replacing s with the program of R, and let S'_1 be the schema obtained from S' by replacing s' with the program of R'. Since s and s' are logic equivalent, S_1 and S'_1 must be equivalent schemas. Lemma 5.2 ensures, therefore, that each predicate symbol which appears in a main iteration statement of S_1 must appear in a main iteration statement of S'_1, and the result follows immediately.

□

We are now in a position to prove the following rather intuitive result:

Lemma 5.6:

Let S and S' be equivalent, conditional-free, RLSA schemas in proper form. Then each iteration statement in S is logic equivalent to some iteration statement in S'.

Proof: For simplicity, we assume that S and S' have only two levels of iteration: the obvious generalization is left to the reader.

Let s be a main iteration statement of S, and let R be its body. Let s' be the main iteration statement of S' which is logic equivalent to s, and let R' be the body of s'. Lemma 5.5 ensures that for each main iteration statement r of R, there is a main iteration statement r' of R' containing the same predicate symbol.

Let S_1 and S'_1 be the schemas constructed as in the proof of Lemma 5.5. According to Lemma 5.4, r and r' must be logic equivalent statements in these schemas, implying that precisely the same tests are made at each statement during the first executions of R and R' in any pair of finite, consistent, free executions of S and S'.

Let S_2 be the schema obtained from S by replacing s with two copies of the program of R, and let S'_2 be obtained similarly from S'. From the preceding argument, we have that the first instance in S_2 of each main iteration statement of R must be logic

equivalent to the first instance in S'_2 of the statement containing the same predicate symbol. Hence, we may change the predicate symbol in each such pair of statements to some new symbol which does not appear elsewhere in the schemas, without affecting the equivalence of S_2 and S'_2. Since the resultant schemas are single appearance schemas, we can apply Lemma 5.4 to demonstrate that the second occurance in S_2 of the statement r is logic equivalent to the second occurance in S'_2 of statement r', implying that the same tests are made at r and r' during the first and second executions of R and R' in any pair of finite, consistent, free executions of S and S'.

For any $n > 0$, we need only apply this argument n times to demonstrate that, for such executions, the same tests are made at r and r' during the first n executions of R and R'; hence, r and r' are logic equivalent.

□

An immediate consequence of the lemma is that predicate symbols of such schemas S and S' must be similarly "nested" in each schema, i.e. if symbol p is in the body of the statement in S containing symbol q, then p must be in the body of the statement in S' containing q.

Suppose we wish to decide the equivalence of RLSA schemas S and S', and suppose neither schema contains main conditional statements. From a previous lemma, we know that if S contains a main trap which has predicate

symbol p and test variable x, then S' can be equivalent to S only if S' contains a similar main trap. Suppose that this is the case:

Let S_p be the schema obtained from S by adding a new variable symbol v_p to the schema and following each statement of the form

$$x \leftarrow f(x)$$

in the portion of S preceding the trap with the statement

$$v_p \leftarrow f(v_p)$$

and then deleting the trap. (Intuitively, v_p is used during an execution of S_p to "record" the value of x which would have been tested at the trap during the corresponding execution of S.) Let S'_p be the schema obtained in a similar manner from S'.

We know from the proof of Lemma 5.3 that if S and S' are equivalent, the value tested at the trap in S must be the same as that tested at the trap in S' during any pair of consistent, finite executions of the schemas. Hence, S and S' are equivalent if and only if S_p and S'_p are equivalent, and it is clear that by repeating the construction for each corresponding pair of traps in the schemas, we can reduce the equivalence problem for S and S' to that for similar schemas which have no main traps.

Lemma 5.7:

Let S and S' be conditional-free RLSA schemas, and let $k > 0$ be the maximum of their sizes. Then we may construct from S and S' two pairs of conditional-free RLSA schemas of size no greater than k-1, such that

S and S' are equivalent if and only if each of the pairs comprises equivalent schemas.

Proof: We may assume that S and S' have the same variable symbols and that their predicate symbols are similarly nested, otherwise we can immediately conclude that they are not equivalent. Without loss of generality, we may also assume that the schemas are without main traps.

Let s be the last iteration statement in the program of S, and let s' be the statement in S' containing the same predicate symbol as s. Let S_0, S'_0, S_1, and S'_1 be the schemas constructed from S and S' as in the proof of Theorem 5.1, and suppose that S_0 is equivalent to S'_0 and S_1 is equivalent to S'_1:

Every statement which follows s' in S' must be logic equivalent in S'_0 to a corresponding statement in S_0, and must be logic equivalent in S'_1 to the same statement in S_1. Clearly, this can be the case only if each variable modified at s' is passive in all statements in S' which follow s'. Now, let E and E' be arbitrary consistent, finite, free executions of S and S', and let x be any variable symbol of the schemas:

We may write the value associated with x at the conclusion of E as $\gamma \cdot \beta \cdot \alpha$, where α is the value associated with x just prior to the first execution of statement s, $\beta \cdot \alpha$ is the value associated with x immediately following the last execution of statement s, and γ is the fixed portion of the value due to the

assignment statements affecting x which follow statement s in the schema. We may write the value associated with x at the conclusion of E' as $\gamma' \cdot \beta' \cdot \alpha'$, where α', β', and γ' are defined in a manner similar to that above, although γ' is not fixed but rather depends in general on the particular execution E' being considered. Since each variable modified at s' is passive in all statements following s', it must be the case that $\gamma \cdot \alpha = \gamma' \cdot \alpha'$, and thus that γ' and α' are completely determined by α. Also, it must be the case that β' is completely determined by β since every test made during an execution of the body of s must also be made during the corresponding execution of the body of s', else by Lemma 5.6 we contradict the equivalence of S_1 and S'_1.

Thus, there must exist functions f_{x_1}, f_{x_2}, and f_{x_3} such that for all executions E and E' as above, $\alpha' = f_{x_1}(\alpha)$, $\beta' = f_{x_2}(\beta)$, and $\gamma' = f_{x_3}(\alpha)$. As noted above, the portion of β' due to a particular execution of s' during E' depends only on that portion of β due to the corresponding execution of s during E, and hence concatenation distributes over f_{x_2}. The lemma then follows from the Elimination Theorem:

S and S' are equivalent if and only if S_0 and S'_0, and S_1 and S'_1, are pairwise equivalent.

□

Thus we have:

Theorem 5.2:

Let S and S' be conditional-free RLSA schemas. Then it is decidable whether S and S' are equivalent.

Proof: We note that equivalence is trivially decidable for schemas of size 0. The theorem follows immediately from the previous lemma by induction on the maximum of the sizes of S and S'.

□

Theorem 5.2 can hardly be considered a surprising result, and the reader will no doubt have observed that there are rather more direct approaches to this particular problem than that which we have presented. (In particular, Lemma 5.6 can be proved fairly simply without utilizing Lemmas 5.4 and 5.5, although our efforts are hardly wasted since these lemmas are needed in the next section.)

The proof presented, however, illustrates the major steps in the proofs of our other cases:

(i) The transformation of schemas into a form facilitating the identification of logic equivalent statements.

(ii) The demonstration that such statements exist in pairs of equivalent schemas.

(iii) The application of the Elimination Theorem to reduce the equivalence of a pair of schemas to that of "smaller" schemas.

In the present case, all three steps are relatively straightforward. In the next case, however, step (i) is complicated by the presence of conditional statements in the schemas, while in the third case steps (ii) and (iii) are complicated by the multiple appearance of predicate symbols. The major steps, however, are the same in each proof.

5.5.2 Decidability of Equivalence for FRLSA Schemas.

In this section we demonstrate that equivalence is decidable for the class of free, restricted location, single appearance schemas (FRLSA schemas).

We begin by showing that essentiality is decidable in such schemas:

Lemma 5.8:

Let S be an FRLSA schema. Then for each predicate symbol p in S, it is decidable whether or not p is essential.

Proof: Since the schema is free, it is clear that p is an essential symbol if it appears in an iteration statement in S. Suppose, therefore, that p appears in a conditional statement. We claim that p is essential if and only if the true and false subschemas of this statement are not equivalent:

Let s be the conditional statement, and suppose that its subschemas are not equivalent. Then we can find partial executions E_t and E_f of S such that the executions end immediately after the first execution of a subschema of s, the executions conflict only with respect to the outcome of the test made at s, and the value associated with some variable symbol x of S is different at the end of each of the executions. We complete the execution E_t in such a way that the outcome of each test subsequently made at a WHILE statement is false, the outcome of each test subsequently made at an UNTIL statement is true, and the outcome of each test subsequently made at a conditional statement is, say, true in each

case. We complete the execution E_f in a similar manner. Since S is a single appearance schema, these complete executions conflict only with respect to the outcome of the test made at s. But since precisely the same sequence of statements is executed after s in each case, the value of x at the end of the executions must still be different. Hence, p is an essential symbol if and only if the subschemas of s are not equivalent.

But it is clearly decidable whether the subschemas of s are equivalent:

If either subschema contains an iteration statement, the subschemas cannot be equivalent since the other subschema cannot contain an iteration statement with the same predicate symbol. If neither subschema contains an iteration statement, their equivalence is trivially decidable since there are only a finite number of distinct free executions of the schemas.

Hence, it is decidable whether p is essential.

□

We say that a schema is <u>reduced</u> if it contains no inessential symbols. We note that a conditional statement containing an inessential symbol may be replaced in a schema with the program of either of its subschemas, and thus we have:

Corollary 5.9:

Let S be a FRLSA schema. Then we may construct from S a reduced FRLSA schema equivalent to S. __/

It may be worth mentioning here that the decision not to permit the designation of certain schema variables as "output" variables is rewarded with the relative simplicity of the proof of Lemma 5.8. While the result is still true for schemas with specified output variables (it was established by the author in [22] for a class of schemas somewhat more general that the FRLSA schemas), the proof becomes rather complex in that case, since we must show that the final value of some output variable, rather than some arbitrary variable, is dependent on the outcome of a test made at a conditional statement. Since this dependency may be quite indirect, the decision procedure for such schemas is quite complicated.

We use some of the results of the previous section to prove the following lemma:

Lemma 5.10:

Let S and S' be reduced, equivalent FRLSA schemas. Then predicate symbol p appears in a conditional/WHILE/UNTIL statement in S if and only if it appears in a conditional/WHILE/UNTIL statement in S'. Moreover, for each iteration statement in S there is a logic equivalent iteration statement in S', and if s and s' are such iteration statements, the sets of predicate symbols appearing in their schemas are the same.

Proof: Suppose that predicate symbol p appears in a main conditional statement s of S. Since p is an essential symbol, it must appear in some statement s' in S'. Moreover, s' must be a conditional statement:

Suppose otherwise. In particular, suppose that s' is a WHILE statement. Let S_o be the schema obtained from S by replacing s with the program of its true subschema, and let S'_∞ be the "schema" obtained from S' by replacing s' with infinitely many copies of its body. S_o must be equivalent to S'_∞, but since S' is free, at least one of its variables (namely the test variable of s') is modified at s', and thus S_o cannot be equivalent to S'_∞. Hence, s' must be a conditional statement. We note also that since p is essential, a productive test may be made at s' each time it is executed. If S and S' are to be equivalent, therefore, s' must be a main conditional statement of S'.

We are thus assured that for each main conditional statement in S, there is a main conditional statement in S' with the same predicate symbol. Let S_t and S_f be the schemas obtained from S by replacing each main conditional statement with the program of its true subschema and its false subschema, respectively. Let S'_t and S'_f be obtained in a similar manner from S'. Clearly, S_t must be equivalent to S'_t, and S_f must be equivalent to S'_f. Applying Lemmas 5.2 and 5.4 to each pair of schemas, we establish the desired result for the symbols occuring

in main statements of the schema. We need only employ the methods of Lemmas 5.5 and 5.6 to establish the result for all symbols.

□

Again, the preceding result is highly intuitive. We note, however, that the result is hardly less intuitive, though demonstrably false, if we remove the restriction of freeness from the schemas.

We now show that a "weak" form of logic equivalence must hold between certain main conditional statements of equivalent, reduced, FRLSA schemas. The following definition is useful:

Let r be a main conditional statement in an iteration schema S. Then r is a final main conditional statement if, whenever r is executed during an execution of S, no main conditional is executed after r.

Lemma 5.11:

Let S and S' be equivalent, reduced, FRLSA schemas, and let r be a final main conditional statement of S. Let E and E' be consistent, finite, free executions of S and S', respectively. Then a test τ is made at r during E only if τ is made during E'.

Proof: Suppose that such a test is made during E. If either subschema of r contains an iteration statement s, then from Lemma 5.10 s can execute during E only if the statement logic equivalent to s executes during E'. Hence, τ cannot be free with respect to E'.

Suppose that neither subschema of r contains an iteration

statement, i.e. suppose that both subschemas are composed solely of assignment statements. Let x be the test variable of r, and let r' denote the main conditional in S' containing the same predicate symbol as r. Since S and S' are equivalent and reduced, the test variable of r' must also be x and no iteration statement which modifies x can precede r' in the program of S' unless its logic equivalent counterpart precedes r in the program of S.

Let E_o be the execution obtained from E by choosing outcome <u>false</u> for each test made at a WHILE statement after the execution of r, and outcome <u>true</u> for each test made at an UNTIL statement; let E'_o be a consistent execution of S'. The test τ is made during E_o, and since the predicate symbol in r is essential and the subschemas of r consist solely of assignment statements, the value associated with some variable symbol y after the execution of r during E_o must depend on the outcome of τ. Also, since an assignment statement is executed after the execution of r in E_o just if the assignment statement is a main statement of S, the value associated with y at the <u>conclusion</u> of E_o must also depend on the outcome of τ. Since S and S' are equivalent schemas, test τ must be made during E'_o and hence during E'.

□

We may apply the lemma recursively to a pair of such schemas S and S' to obtain:

Corollary 5.12:
Let S and S' be as in Lemma 5.11. Let s be a main conditional of S, and let s' be the main conditional of S' with the same predicate symbol. Then during any pair of finite, consistent, free executions of S and S', either no test is made at s or s', or the same test is made at both statements. ⌟

A consequence of the corollary is that the equivalence problem for FRLSA schemas reduces to that for FRLSA schemas with no main conditional statements:

Let S and S' be reduced FRLSA schemas, and suppose that for each main conditional in S there is a main conditional in S' with the same predicate symbol and test variable. (If such is not the case, we may conclude from Lemma 5.10 that S and S' are not equivalent.) Let r be a final main conditional of S, and let r' be the corresponding main conditional of S'. We construct from S and S' a pair of schemas S_o and S'_o, as follows:

We add a new variable symbol x_r to each schema to record the values tested at r and r' during executions of the schemas, in the same manner as when eliminating main traps from the schemas of Section 5.5.1. If r is in a subschema of some other main conditional of S, we replace this conditional with the program of the subschema and replace

the corresponding conditional in S' with the program of its corresponding subschema. This is repeated until r is no longer in a subschema of a main conditional of S, and then r and r' are replaced with the programs of one of their subschemas, say their true subschemas. The resultant schemas are S_o and S'_o.

We construct a second pair of schemas S_t and S'_t from S and S' by replacing r and r' with the programs of their true subschemas, and reducing the resultant schemas; a third pair of schemas S_f and S'_f are constructed in a similar manner, replacing r and r' with the programs of their false subschemas.

Intuitively, the equivalence of S_o and S'_o ensures that a test made at r during an execution of S must also be made at r' during any consistent execution of S'. Of course, this presupposes that no test made at a conditional statement of S whose subschema contains r will have an outcome different from that of the test made at its counterpart in S', but from Corollary 5.12, this is the case if S_t is equivalent to S'_t, and S_f is equivalent to S'_f. Hence, S and S' are equivalent if and only if S_o and S'_o, S_t and S'_t, and S_f and S'_f, are pairwise equivalent. Since each of these schemas contains at least one fewer main conditional than S and S', the constructions may be repeated to reduce the equivalence of S and S' to that of a finite number of pairs of schemas without main conditionals.

The proof of the following lemma is virtually identical to that of Lemma 5.7 and is left to the reader:

Lemma 5.13:

Let S and S' be FRLSA schemas without main conditionals, and let $k > 0$ be the maximum of their sizes. Then we may construct from S and S' two pairs of FRLSA schemas of size no greater than k-1, such that S and S' are equivalent if and only if each of the pairs comprises equivalent schemas. _/

We have observed that the equivalence of a pair of FRLSA schemas can be reduced to that of a finite number of pairs of FRLSA schemas without main conditionals. We note that the latter schemas are of the same size as the originals, and thus we have:

Theorem 5.3:

Let S and S' be FRLSA schemas. Then it is decidable if S and S' are equivalent.

Proof: From the previous lemma, by induction on the size of S and S'.

□

5.6 Decidability of Equivalence for FILCF Schemas.

In the previous section we dealt with schemas in which predicate symbols appeared only once. Finding logic equivalent statements in equivalent schemas was quite easy, therefore, because we knew precisely which pairs of statements to examine — those containing the same predicate symbols.

In this section we consider the equivalence problem for free, independent location, conditional-free schemas (FILCF schemas). As might be expected, the proof that logic equivalent statements exist in equivalent pairs of such schemas is not quite as trivial as it was in the previous cases:

Lemma 5.14:

Let S and S' be FILCF schemas, and let s be the last iteration statement in the program of S. Then if S and S' are equivalent, there exists a statement s' in the program of S' such that s and s' are logic equivalent.

Proof: Suppose that S and S' are equivalent. Let E be a free execution of S such that TESTS(s,E) contains infinitely many elements, and let E' be a consistent execution of S'. (We note that since E is non-terminating and S and S' are equivalent schemas, E' must be a non-terminating execution.) Clearly, every element of TESTS(s,E) must be an element of TESTS*(S',E'), since if some $\tau \in$ TESTS(s,E) were not in TESTS*(S',E'), we could change the outcome of τ during E to obtain a terminating execution of S consistent with the non-terminating execution E', contradicting

the equivalence of S and S'. Also, all but finitely many of the elements of TESTS(s,E) must be in TESTS(s',E') for some statement s' in S':

Suppose otherwise. Then there must exist statements s_1 and s_2 in schema S' such that TESTS(s_1,E') and TESTS(s_2,E') each contain infinitely many elements of TESTS(s,E). (For simplicity we shall assume that s_1 and s_2 are the only such statements.) It must be the case that one of the statements, say s_2, is in the body of the other. Let τ be any test in TESTS(s,E) $\cap$ TESTS(s_2,E'). If we change the outcome of τ during E, we obtain a terminating execution E'' of S; also, since S is an independent location schema, no value which is longer than the first component of τ and ends with the same symbol is tested during E''. Suppose we also change the outcome of τ during execution E': the first test made at statement s_1 after τ is made at s_2 must be free with respect to E'' since the value tested must be longer that that tested in τ and must end with the same symbol. If s_1 is a WHILE statement, we choose the outcome of this test to be <u>true</u>, as we do for each subsequent test made at s_1 (each of which must also be free with respect to E''). If s_1 is an UNTIL statement, we choose a succession of <u>false</u> outcomes. In either case, the resultant execution is consistent with E'' but is non-terminating, contradicting the equivalence of S and S'. Hence, there must exist a statement s' such that all but finitely many elements

of TESTS(s,E) are in TESTS(s',E').

We now show that s' must be in the program of S':

Let x be the test variable of statements s and s', and suppose that s' is not in the program of S', i.e. suppose that s' is in the body of some iteration statement r_0 in the program of S'. Clearly, no statement r which follows r_0 in the program can have test variable x, otherwise we could obtain from E a terminating execution of S and from E' a consistent, non-terminating execution of S' by changing the outcome of any test $\tau \in (TESTS(s,E) \cap TESTS(s',E'))$ as above, using s' in place of s_2 and r in place of s_1. Similarly, the statement r_0 itself cannot have test variable x. Let τ' be the first test made at r_0 with the property that during the subsequent execution of r_0's body, some test $\tau'' \in TESTS(s,E)$ is made at s'. Let E_1 be an execution of S consistent with E except for the outcome of test τ'' (if τ'' is made during E), and such that $TESTS(s,E_1)$ is infinite. Let E'_1 be an execution of S' consistent with E_1 and consistent with the portion of E' preceding the execution of r_0. If we apply the arguments of the preceding paragraph, we have that there exists a statement s'_1 in schema S' such that all but finitely many of the elements of $TESTS(s,E_1)$ are in $TESTS(s'_1,E'_1)$. Moreover, s'_1 must be in the body of some iteration statement r_1 which follows r_0 in the program of S', since E and E'_1 are the same prior to the execution of r_0, and s'_1 itself cannot be in the program of S'.

But if we repeat this argument ad infinitum, we can show that there must exist an infinite sequence of statements $r_0, r_1, r_2, \ldots$ which follow one another in the program of S', an impossibility since the program must be finite. Hence, the statement s' must be in the program of S' (and must in fact be the last iteration statement in the program with test variable x).

The logic equivalence of the statements is now easily demonstrated:

We note that for any pair of finite, consistent, free executions E_o of S and E'_o of S', the last element of TESTS(s,E_o) must be the same as the last element of TESTS(s',E'_o). If this were not the case, we could change the outcome of whichever test had the longer first component, or either test if the components were of equal length, without disrupting the consistency of the executions. This would cause another test to be made at the corresponding statement and since this test, and all subsequent tests made at the statement, would be free with respect to the execution of the other schema, we could permit this execution to diverge while still remaining consistent with the other, thus contradicting the equivalence of S and S'.

Hence, the last elements of TESTS(s,E_o) and TESTS(s',E'_o) must be the same for any such E_o and E'_o. But since the schemas are free, this can be the case only if TESTS(s,E_o) = TESTS(s',E'_o):

Suppose otherwise. Then there must be a test τ in one of the sequences, say TESTS(s,E_o), which is not in the other. Since τ

must be made during E'_0, it must be the case that $\tau \in TESTS(r,E'_0)$ for some statement r whose execution precedes that of s' during E'_0. Let E_1 be the execution of S conflicting with E_0 only with respect to the outcome of τ, and let E'_1 be an execution of S' which is consistent with E_1 and is consistent with the portion of E'_0 which precedes the making of test τ. It must be the case that $\tau \in TESTS(r,E'_1)$, but since τ is the last element of $TESTS(s,E_1)$ it must also be the last element of $TESTS(s',E'_1)$, contradicting the freeness of S'.

We have, therefore, that $TESTS(s,E_0) = TESTS(s',E'_0)$ for any pair of finite, consistent, free executions E_0 and E'_0, and thus s and s' are logic equivalent.

□

Lemma 5.15:

Let S and S' be equivalent FILFC schemas of size greater than 0. Let s be the last iteration statement in the program of S, and let s' be the logic equivalent iteration statement in S'. Then each variable which is modified at s' is passive in all statements which follow s' in the program of S'.

Proof: Suppose otherwise. In particular, suppose that a variable x is modified at s' and active in some statement s'' which followe s'. (For simplicity, we assume that x is the only such variable and s'' the only such statement — the generalization is tedious

but straightforward.) We can assume without loss of generality that x is the test variable of s", for if it is instead the test variable of some statement in the body of s", we merely restrict attention in the ensuing discussions to executions of S' during which this statement is executed.

Since x is modified at statement s' it is clearly the case that x is modified at s, since otherwise the value of x would be independent of the number of times s were executed during a free execution of S, while the value of x grows, in general, in proportion to the number of times the statement s' is executed during a free execution of S'.

Also, it must be the case that x is active in statement s:

Suppose otherwise. Let E and E' be consistent, finite, free executions of S and S' such that the value associated with x after the last execution of s' during E' is longer than that associated with x immediately prior to the first execution of s during E. Since we are assuming that x is passive in s, the first test made at s" during E' must be free with respect to E, as must any subsequent tests made at s". By choosing <u>true</u> outcomes for all of these tests, if s" is a WHILE statement, or <u>false</u> outcomes for all of these tests, if s" is an UNTIL statement, we obtain a non-terminating execution of S' consistent with E, which contradicts the equivalence of S and S'. Hence, x must be active in s. But then s must also

be active in s':

Suppose otherwise. Let R be the body of s, and let s_x be a statement with test variable x in R. Let E_x be a free execution of S in which s_x is executed infinitely often during the first execution of R. Let E'_x be an execution of S' consistent with E_x in which the body of s' is executed exactly twice. Since S and S' are equivalent schemas, E'_x must be a non-terminating execution. In fact, it must be the case that TESTS(s",E'_x) is infinite and that all but finitely many of its elements are in TESTS(s_x,E_x). Let τ be in (TESTS(s_x,E_x) $\cap$ TESTS(s",E'_x)), and let $\hat{E}_x$ be an execution of S which is consistent with E_x except for the outcome of τ and which has the property that s_x is executed infinitely often during the second execution of R in $\hat{E}_x$. Then since x is passive in all statements following s" in the program of S', we can find a finite execution $\hat{E}'_x$ of S which is consistent with $\hat{E}_x$:

The execution $\hat{E}'_x$ will be consistent with E'_x until the test τ is made. Since each statement subsequently executed has a test variable other than x, no more than finitely many tests need be made at any statement before a test is made which is free with respect to $\hat{E}_x$. We choose the outcome of such a test to be <u>true</u> if it is made at an UNTIL statement, or <u>false</u> if it is made at a WHILE statement. The resultant execution is clearly terminating and is consistent with $\hat{E}_x$,

contradicting the equivalence of S and S'. Hence, x must be active in both s and s'.

Let R' be the body of s', and let s'_x be a statement in R' with test variable x. For simplicity, we assume that s'_x is the only statement in R' with test variable x and that s_x is the only statement in R with test variable x; the generalization is again straightforward. Let E' be a free execution of of S' in which s'_x is executed infinitely often during the first execution of R', and let E be a consistent execution of S. No more than finitely many elements of $TESTS(s_x,E)$ may be free with respect to E', otherwise we may terminate the execution of S in a manner consistent with E', contradicting the equivalence of the schemas. Hence by changing the outcome of some test $\tau \in (TESTS(s_x,E) \cap TESTS(s'_x,E'_x))$, we may obtain from E a finite execution E_τ of S such that E_τ is consistent with E except for the outcome of τ, and such that τ is the last element of $TESTS(s_x,E_\tau)$. Let E'_τ be an execution of S' which is consistent with E_τ and is also consistent with the portion of E' prior to the making of τ. We note that the first test made at s" during E'_τ must be free with respect to E_τ, as must each subsequent test made at s", and hence that E'_τ can be chosen so that it is finite, again contradicting the equivalence of S and S'.

Thus no such variable x can exist, and each variable which is modified at s' must be passive in all statements which follow

s' in the program of S'.

□

The proof of the following lemma is again virtually identical to that of Lemma 5.7:

Lemma 5.16:

Let S and S' be FILCF schemas, and let $k > 0$ be the maximum of their sizes. Then we may construct from S and S' two pairs of FILCF schemas of size no greater than k-1, such that S and S' are equivalent if and only if each of the pairs comprises equivalent schemas. __/

Finally, we have:

Theorem 5.4:

Let S and S' be FILCF schemas. Then it is decidable whether or not S and S' are equivalent.

Proof: From Lemma 5.16, by induction on the size of S and S'.

□

5.7 Discussion.

Obviously, the preceding results do not provide answers to all of the questions raised in Chapter IV. We conjecture, however, that techniques similar to those we have presented are applicable to most of the classes of schemas considered in the last paragraph of that chapter, and that equivalence is decidable for each of the classes. For certain of the classes, however, it is not clear that the additional efforts required to establish decidability results are well-spent:

Our motive for studying the conditional-free schemas of the last section, for example, is that such schemas may provide a suitable basis for the study of structured independent location schemas in general, if methods can be developed to remove in some systematic manner the conditional statements from such schemas. It would hardly seem worthwhile, therefore, to expend much effort in extending Theorem 5.4 to non-free schemas, since the equivalence problem for such schemas can be shown reducible to that for free schemas in which conditionals are permitted. Of course, it might be argued that freeness is a rather undesirable restriction since it is not a decidable property of iteration schemas, but it is fairly easy to show (see Appendix B) that freeness <u>is</u> a decidable property of independent location schemas.

A similar comment applies to the result for FRLSA schemas. Again, we hope to be able to apply the result to more general classes of schemas by identifying logic equivalent statements in pairs of schemas and suitably changing the predicate symbols in the statements. (In fact,

Theorem 5.4 can be derived in such a way from Theorem 5.2, although the proof that we have presented is somewhat more direct.) In light of the undecidability results in Chapter IV for restricted location schemas, freeness will likely be a necessary restriction if equivalence is to be decidable for these more general classes. We lose little, therefore, by imposing the restriction now.

A few final words are in order about a class of schemas which does seem worth considering, however, and that is the class of free and conditional-free schemas which are restricted location, rather than independent location. We conjecture that Lemma 5.14 is still valid for such schemas, although the proof is complicated by the fact that the set of tests whose first components end with some given symbol Δ_x need not be made in order of increasing lengths of these components. The remainder of the proof for the independent location schemas is, with quite minor and obvious modifications, applicable to restricted location schemas.

CHAPTER VI:

Independent location program schemas have been studied rather extensively (cf, [12], [16], [19]) because of the schemas' relative simplicity and because their equivalence problems are interchangable with those of a rather interesting class of automata, the multi-tape finite automata defined in [23]. In this chapter, we consider the way in which the equivalence problems for structured independent location schemas relate to those for such schemas in general, and to those for certain classes of the automata.

We show that the weak equivalence problems for structured and non-structured independent location schemas are interchangable, and that both problems are in fact unsolvable. While we are not able to show that the equivalence problem for multi-tape automata is reducable to that for structured independent location schemas, we <u>are</u> able to show that the equivalence problem for multi-tape automata with a single control state reduces to that for such schemas, and that the strong equivalence problem for independent location schemas in general reduces to the problem of deciding whether such automata are equivalent over some subset of their tapes.

6.1 Multi-Tape Finite Automata.

Our treatment of multi-tape automata differs somewhat from that in [16] or [23] since we consider a rather special subclass of the automata in a later section, and we wish the notions developed here to be reasonable for this subclass.

Intuitively, an n-tape automaton M is a finite automaton equipped with n one-way scanning heads, each on its own tape. Associated with M is an advancement function and a transition function which determine, based on the current internal state of M and the n-tuple of symbols being scanned, the tape heads, if any, which are to be advanced to the next symbol, and the internal state of M which is then entered. A particular state of M is designated the initial state of the automaton, another its accepting state, and a third its rejecting state. No transitions are permitted out of these last two states.

Each input tape of M is initially inscribed with a sequence of symbols from some finite tape alphabet followed by a special endmarking symbol \$, beyond which a tape head is not permitted to scan. A computation by M on a set of tapes begins with M in its initial state and each tape head positioned at the leftmost square of its tape, and proceeds until each head is scanning its respective endmarker, at which time a set of tapes is accepted if M is in its accept state or rejected if M is in its reject state. If M is in neither state, or if such a positioning of tape heads never occurs, M is said to diverge on the tapes.

Formally:

An n-tape automaton is a seven-tuple

$$M = (T, Q, q_I, q_a, q_r, f, h)$$

where: T is a finite set of tape symbols, including the special

endmarker \$.

Q is a finite set of control states.

$q_I \in Q$ is the initial state of M.

$q_a \in Q$ is the accepting state of M.

$q_r \in Q$ is the rejecting state of M.

$f: Q \times T^n \to Q$ is the state transition function, a total function satisfying the property that $f(q_a,\varphi) = q_a$ and $f(q_r,\varphi) = q_r$, all $\varphi \in T^n$.

$h: Q \times T^n \to 2^{\{1,2,\ldots,n\}}$, where $2^{\{1,2,\ldots,n\}}$ denotes the power set of $\{1,2,\ldots,n\}$, is the head advancement function, a total function satisfying the property that $h(q,\varphi)$ does not contain i whenever the ith component of φ is \$, all $q \in Q$ and $\varphi \in T^n$.

A configuration of M is a pair (q,Λ), where q is an element of Q and Λ is an n-tuple of strings in T^+. (For such a Λ, we denote by TAIL(Λ) the string of length n whose ith symbol is the last symbol of the ith component of Λ, $1 \le i \le n$.)

A computation by M is a possibly infinite sequence of configurations

$(q_1,\Lambda_1),(q_2,\Lambda_2), \ldots , (q_k,\Lambda_k),(q_{k+1},\Lambda_{k+1}), \ldots$

in which q_1 is the initial state q_I, Λ_1 is a tuple of single symbols, and for all $i > 1$:

(1) $q_i = f(q_{i-1},\text{TAIL}(\Lambda_{i-1}))$

(2) $\Lambda_i = \Lambda_{i-1} \cdot S$, where S is an n-tuple of symbols or nulls in $T \cup \{\lambda\}$, λ the null string, such that for all j, $1 \le j \le n$, the jth component of S is λ iff $j \notin h(q_{i-1},\text{TAIL}(\Lambda_{i-1}))$. (Concatenation is extended to tuples of strings in the obvious manner: if $X = (\delta_1, \ldots , \delta_k)$

and $Y = (\sigma_1, \ldots, \sigma_k)$ are tuples of strings, then $X \cdot Y$ is the tuple $(\delta_1 \cdot \sigma_1, \ldots, \delta_k \cdot \sigma_k)$.)

An n-tuple of strings in T^* is <u>accepted</u> by M if there is a finite computation by M ending with the configuration $(q_a, \Lambda \cdot \$^n)$; it is <u>rejected</u> by M if there is a finite computation ending with the configuration $(q_r, \Lambda \cdot \$^n)$. If Λ is neither accepted nor rejected by M, we say that M <u>diverges</u> on Λ. We note that since there are no transitions leaving q_a or q_r, no Λ may be both accepted and rejected by M.

The <u>language accepted</u> by M, written L(M), is the set $\{\Lambda \mid \Lambda$ is accepted by M$\}$. The <u>language rejected</u> by M, written $\bar{L}(M)$, is the set $\{\Lambda \mid \Lambda$ is rejected by M$\}$.

6.2 Equivalence Problems For Multi-Tape Automata.

Let M and M' be n-tape automata, some $n > 0$. Then M and M' are <u>strongly equivalent</u> if $L(M) = L(M')$ and $\bar{L}(M) = \bar{L}(M')$. M and M' are <u>weakly equivalent</u> if $(L(M) \cap \bar{L}(M')) = \emptyset$ and $(\bar{L}(M) \cap L(M')) = \emptyset$, i.e. if no tuple of strings accepted by M is rejected by M', and vice versa.

Let M and M' be n-tape automata, and let $N = \{i_1, \ldots, i_\ell\}$ be a set of integers between 1 and n, inclusive. Then M and M' are <u>N-restricted equivalent</u> if the set $\{(\delta_1, \ldots, \delta_\ell) \mid$ for some $\Lambda \in L(M)$, δ_{i_j} is the i_jth component of Λ, $1 \leq j \leq \ell\}$ is equal to the set $\{(\delta_1, \ldots, \delta_\ell) \mid$ for some $\Lambda \in L(M')$, δ_{i_j} is the i_jth component of Λ, $1 \leq j \leq \ell\}$, i.e. if M and M' accept the same tuples of strings when attention is restricted to the subset of their tapes designated by N.

6.3 "Equivalence" of Independent Location Schemas and Multi-Tape Finite Automata.

Informally, we consider two models for computation to be equivalent if their equivalence problems are interchangable and if, given an element in one model, we can effectively construct an element in the other which simulates it in some well-defined manner. Luckham, Park, and Paterson have demonstrated that independent location schemas with n variable symbols are equivalent in this sense to finite automata with n tapes. The simulations are straightforward (indeed the simulation of the schemas by the automata is rather trivial), but they are of no particular interest to us here; the reader is referred to [16] for details. We do note here, however, that the weak equivalence and strong equivalence problems for the automata correspond directly to the weak equivalence and strong equivalence problems for the schemas, while the restricted equivalence problem for the automata corresponds to the equivalence problem for independent location schemas with designated output symbols.

6.4 Weak Simulation of Multi-Tape Automata by Structured Independent Location Schemas.

It follows from the discussion in the preceding section that any independent location iteration schema can be simulated by some multi-tape automaton. In this section we demonstrate that an arbitrary multi-tape automaton can be "weakly" simulated by some such schema, i.e. simulated in such a way that each computation by the automaton corresponds to an execution of the schema for a suitably chosen interpretation, and each

terminating execution of the schema corresponds to some computation by the automaton. We are thus able to demonstrate the correspondence of the weak equivalence problems for the automata and the schemas, though we are not able to demonstrate a correspondence of their strong equivalence problems.

6.4.1 The Simulation.

Let $M = (\{s_1,\ldots,s_k,\$\}, \{q_1,\ldots,q_m,q_I,q_a,q_r\}, q_I, q_a, q_r, f, h)$ be an n-tape automaton, for some $n > 0$, as defined in Section 6.1. We show how to construct an independent location iteration schema S_M which weakly simulates the automaton:

S_M will have variable symbols $x_1,\ldots,x_n$, representing the tapes of M. It will have an additional variable symbol y which will be used to record the states entered by M during a simulated computation, and also to record the outcome of the computation (acceptance or rejection) if the computation does not diverge.

S_M will have predicate symbols $p_{s_1},\ldots,p_{s_k},p_\$$, corresponding to the tape symbols of M. It will also have predicate symbols $p_{q_1},\ldots,p_{q_m},p_I$, p_{q_a},p_{q_r}, corresponding to the states of M, and a "number of moves" symbol p_m.

S_M will have function symbol g representing the advancement of a tape head, and symbols a and r denoting acceptance and rejection, respectively.

For notational convenience, we provide simple representations for certain boolean expressions:

For each i, $1 \leq i \leq k$, and each j, $1 \leq j \leq n$, we represent by $P_{s_i}(x_j)$ the expression $(p_{s_i}(x_j) \wedge (\neg(p_{s_1}(x_j) \vee p_{s_2}(x_j) \vee \ldots \vee p_{s_{i-1}}(x_j) \vee p_{s_{i+1}}(x_j) \vee \ldots \vee p_{s_k}(x_j) \vee p_{\$}(x_j)))$.

Similarly, for each j, $1 \leq j \leq n$, we represent by $P_{\$}(x_j)$ the expression $(p_{\$}(x_j) \wedge (\neg(p_{s_1}(x_j) \vee \ldots \vee p_{s_k}(x_j)))$.

For each i, $1 \leq i \leq m$, we represent by P_{q_i} the expression $(p_{q_i}(y) \wedge (\neg(p_{q_1}(y) \vee p_{q_2}(y) \vee \ldots \vee p_{q_{i-1}}(y) \vee p_{q_{i+1}}(y) \vee \ldots \vee p_{q_m}(y) \vee p_{q_I}(y) \vee p_{q_a}(y) \vee p_{q_r}(y)))$, and define expressions for P_{q_I}, P_{q_a}, and P_{q_r} in a similar manner.

Let $\varphi_1, \varphi_2, \ldots, \varphi_{(k+1)^n}$ be an enumeration of the strings of length n over the tape alphabet of M. Then for each j, $1 \leq j \leq (k+1)^n$, we represent by P_{φ_j} the expression $(P_{s_{j_1}}(x_1) \wedge P_{s_{j_2}}(x_2) \wedge \ldots \wedge P_{s_{j_n}}(x_n))$, where $s_{j_1} \cdot s_{j_2} \cdot \ldots \cdot s_{j_n} = \varphi_j$.

We adopt a shorthand notation for certain sequences of assignment instructions, as follows:

Let $N = \{i_1, \ldots, i_\ell\}$ be a set of integers between 1 and n, inclusive. Then we denote by $'x_N \leftarrow g(x_N)'$ the sequences of instructions $'x_{i_1} \leftarrow g(x_{i_1})', \ldots, 'x_{i_\ell} \leftarrow g(x_{i_\ell})'$.

The subschemas of S_M are as follows:

The empty schema S_E and the divergent schema $S_\uparrow$, as defined in Chapter IV.

For each i, $1 \leq i \leq (k+1)^n$, and each j, $1 \leq j \leq m$, the subschema $S_{i,j}$ with program:

$x_{h(q_j,\varphi_i)} \leftarrow g(x_{h(q_j,\varphi_i)})$

$y \leftarrow g(y)$

IF $P_{f(q_j,\varphi_i)}$ THEN S_E ELSE $S_\uparrow$

For each i, $1 \le i \le (k+1)^n$, the subschema $S_{i,I}$ with program:

$x_{h(q_I,\varphi_i)} \leftarrow g(x_{h(q_I,\varphi_i)})$

$y \leftarrow g(y)$

IF $P_{f(q_I,\varphi_i)}$ THEN S_E ELSE $S_\uparrow$

For each i, $1 \le i < (k+1)^n$, and each j, $1 \le j \le m$, the subschema $R_{i,j}$ with program:

IF $(P_{q_j} \wedge P_{\varphi_i})$ THEN $S_{i,j}$ ELSE $R_{i+1,j}$

For each j, $1 \le j < m$, the subschema $R_{(k+1)^n,j}$ with program:

IF $(P_{q_j} \wedge P_{\varphi_{(k+1)^n}})$ THEN $S_{(k+1)^n,j}$ ELSE $R_{1,j+1}$

The subschema $R_{(k+1)^n,m}$ with program:

IF $(P_{q_m} \wedge P_{\varphi_{(k+1)^n}})$ THEN $S_{(k+1)^n,m}$ ELSE $R_{1,I}$

For each i, $1 \le i < (k+1)^n$, the subschema $R_{i,I}$ with program:

IF $(P_{q_I} \wedge P_{\varphi_i})$ THEN $S_{i,I}$ ELSE $R_{i+1,I}$

The subschema $R_{(k+1)^n,I}$ with program:

IF $(P_{q_I} \wedge P_{\varphi_{(k+1)^n}})$ THEN $S_{(k+1)^n,I}$ ELSE $S_\uparrow$

The schema S_M is then:

IF P_{q_I} THEN S_E ELSE $S_\uparrow$

WHILE $p_m(y)$ DO $R_{1,1}$

IF $(P_\$(x_1) \wedge \ldots \wedge P_\$(x_n))$ THEN S_E ELSE $S_\uparrow$

IF $P_{q_a}(y)$ THEN 'y ← a(y)' ELSE S_E

IF $P_{q_r}(y)$ THEN 'y ← r(y)' ELSE S_E

Let I be a free interpretation for S_M, and for each predicate symbol p of S_M let Π_p denote the predicate assigned to p by I.

For each i, $1 \le i \le n$, let ℓ_i be the least integer > 0 such that $\Pi_{p_\$}(g^{\ell_i}\cdot\Delta_{x_i})$ is <u>true</u>, and let ℓ_y be the least integer such that $\Pi_{p_m}(g^{\ell_y}\cdot\Delta_y)$ is <u>false</u>. We say that I is a <u>reasonable</u> interpretation for S_M if exactly one of the predicates $\Pi_{p_{s_1}}, \Pi_{p_{s_2}}, \ldots, \Pi_{p_{s_k}}, \Pi_{p_\$}$ is <u>true</u> at each element of $\{g^i\cdot\Delta_{x_j} \mid 0 \le i \le \ell_j, 1 \le j \le n\}$, and exactly one of the predicates $\Pi_{p_{q_1}}, \Pi_{p_{q_2}}, \ldots, \Pi_{p_{q_m}}, \Pi_{p_{q_I}}, \Pi_{p_{q_a}}, \Pi_{p_{q_r}}$ is <u>true</u> at each element of $\{g^i\cdot\Delta_y \mid 0 \le i \le \ell_y\}$.

Each interpretation I which is reasonable for S_M defines an n-tuple Λ of strings over the alphabet of M in a straightforward manner: the ith symbol in the jth string, $1 \le j \le n$, $1 \le i \le \ell_j$, is symbol s if and only if $\Pi_{p_s}(g^{i-1}\cdot\Delta_{x_j})$ is <u>true</u>.

Each such interpretation I for S_M defines a sequence of states of M in a similar manner: the ith state in the sequence, $1 \le i \le \ell_y$, is q if and only if $\Pi_{p_q}(g^{i-1}\cdot\Delta_y)$ is <u>true</u>.

The reader may verify that S_M diverges under all unreasonable interpretations, and converges for the reasonable interpretation I if and only if the sequence of states defined by I is consistent with a non-divergent computation of M on Λ. If the execution of S_M converges, the symbol a or r is prefixed to the value of symbol y, according as the last state of M in the simulated computation is q_a or q_r.

Thus, S_M weakly simulates the automaton M in the manner described previously.

6.4.2 Weak Equivalence: A Reducibility.

As noted, the schema S_M constructed in the preceding section diverges under all unreasonable interpretations, and diverges under reasonable interpretations which define input strings on which M diverges. But the schema may also diverge under reasonable interpretations which define input strings on which M does <u>not</u> diverge, if the sequence of states defined by the interpretation does not correspond to the sequence of states entered by M during its computation on the strings. This behavior is an inherent feature of the simulation, and is in fact the feature which makes the simulation "weak" (and thus precludes us from reducing the strong equivalence problem for the automata to that for the schemas).

We note also that if the execution of S_M terminates for some interpretation I, the final value associated with symbol y for the execution will depend on the length of M's computation on the input strings defined by I. Since the length of the computation made by an

equivalent automaton on the same set of strings may be quite different from that made by M, it would seem that the simulation is not suitable even for a reduction of the weak equivalence problem. Fortunately, the following lemma implies that such is not the case:

Lemma 6.1:

Let M be an n-tape automaton, for some $n > 0$. Then we may construct from M an equivalent n-tape automaton M' such that precisely one tape head is advanced at each step of any convergent computation by M.

Proof: Let Q and T be the states and tape symbols, respectively, of M. Let f be the transition function and h the head advancement function of M. We first modify M so that no more than a single head is advanced during a step of any computation by the automaton:

Let q be a state in Q and let φ be an element of T^n such that $h(q,\varphi) = \{j_1,\ldots,j_m\}$ for some $m > 1$. We add to Q new states q', q'', ... , $q^{(m-1)}$ and extend f and h to these new states so that $f(q^{(i)},\delta) = q^{(i+1)}$ and $h(q^{(i)},\delta) = \{j_{i+1}\}$, for all $\delta \in T^n$ and all i, $1 \leq i \leq m-1$. For each $\delta \in T^n$, we define $f(q^{(m-1)},\delta)$ to be the state $f(q,\varphi)$ and we define $h(q^{(m-1)},\delta)$ to be $\{j_m\}$. Finally, we redefine $f(q,\varphi)$ to be state q' and $h(q,\varphi)$ to be $\{j_1\}$. The procedure is repeated for any additional arguments for which the value of h is a set of cardinality greater than one, and the resultant automaton has the desired property.

Assume now that no more than one head is advanced during any step in a computation by the automaton. Whenever we have $f(q,\varphi) = q'$ for some states q and q' and some $\varphi \in T^n$ such that $h(q,\varphi) = \emptyset$ and $h(q',\varphi) \neq \emptyset$, we redefine $f(q,\varphi)$ to be $f(q',\varphi)$ and redefine $h(q,\varphi)$ to be $h(q',\varphi)$. This procedure is repeated as long as such q,q' and φ can be found. The resultant automaton is M'.

□

Thus we have:

Theorem 6.1:

The weak equivalence problem for multi-tape finite automata reduces to the weak equivalence problem for independent location iteration schemas.

Proof: Let M_1 and M_2 be n-tape automata for some $n > 0$, and let M'_1 and M'_2 be the automata constructed from M_1 and M_2 as in the preceding lemma. Let $S_{M'_1}$ and $S_{M'_2}$ be the simulating schemas constructed from M'_1 and M'_2 as in Section 6.4.1. Then M and M' are weakly equivalent if and only if $S_{M'_1}$ and $S_{M'_2}$ are weakly equivalent.

□

As we shall see in the next section, Theorem 6.1 is a more interesting result than it seems at first glance, since the weak equivalence problem for the automata can be shown unsolvable.

6.5 Undecidability of Weak Equivalence for Independent Location Schemas.

In this section we demonstrate that weak equivalence is undecidable for multi-tape finite automata and hence, according to Theorem 6.1, for independent location iteration schemas.

The following result was implied in [12] and demonstrated explicitly in [19]. The proof given here is essentially that in the latter paper.

Lemma 6.2:

The inclusion problem for multi-tape finite automata is unsolvable. That is, it is recursively undecidable whether $L(M) \subseteq L(M')$ for arbitrary n-tape automata M and M'.

Proof: Let C be the Post Correspondence Problem defined in Section 4.1. It is a trivial matter to construct a 2-tape automaton M such that $L(M) = \{(\kappa, \omega) \mid \kappa = \bar{i}_1 \# \bar{i}_2 \# \ldots \# \bar{i}_\ell$, where # is some special symbol, and for each j, $1 \leq j \leq \ell$, $\bar{i}_j$ is a symbol denoting integer i_j, $1 \leq i_j \leq k$; and $\omega = \omega_{i_1} \cdot \omega_{i_2} \cdot \ldots \cdot \omega_{i_\ell}\}$. Also, we can construct another 2-tape automaton M' such that $L(M') = \{(\kappa, \hat{\gamma}) \mid \kappa$ is as in L(M) and $\hat{\gamma}$ is any word other than $\gamma_{i_1} \cdot \gamma_{i_2} \cdot \ldots \cdot \gamma_{i_\ell}\}$. Clearly, C has a solution if and only if $L(M) \not\subseteq L(M')$, and hence the decidability of inclusion for multi-tape automata implies the solvability of the unsolvable Post's Correspondence Problem.

□

We now reduce the inclusion problem for multi-tape finite automata to the weak equivalence problem for the automata.

Lemma 6.3:

Let M be an n-tape automaton, for some $n > 0$. Then we can construct from M an n-tape automaton M' such that $L(M) = L(M')$ and M' rejects no input.

Proof: Informally, we add a new state q to the states of M, and provide transitions from q back to q for each length n string from M's alphabet. Each transition into the rejecting state of M is replaced with a transition into this new state, and the resultant automaton is M'.

□

The following lemma is derived immediately from Lemma 6.1:

Lemma 6.4:

Let M be an n-tape automaton, for some $n > 0$. Then we can construct from M an n-tape automaton M' such that $L(M) = L(M')$ and M' rejects any input which is not accepted.

Proof: We construct from M the automaton M' of Lemma 6.1. We than add a new state q to this machine, and redefine $f(q_i,\varphi)$ to be q whenever $h(q_i,\varphi) = \emptyset$, where f and h are the transition and advancement functions of M', q_i is any state in M', and φ is any length n string of tape symbols. We extend f and h to state

q as follows:

If φ is a string of length n other than $\$^n$, we define $f(q,\varphi)$ to be q and $h(q,\varphi)$ to be $\{i \mid$ the ith component of φ is not \$, $1 \leq i \leq n\}$. If φ is $\$^n$, then $f(q,\varphi)$ is q_r, where q_r is the rejecting state of M', and $h(q,\varphi)$ is the empty set. The resultant automaton is M''.

□

From these lemmas we obtain:

Theorem 6.2:

The inclusion problem for multi-tape finite automata is reducible to the weak equivalence problem for the automata.

Proof: Let M_1 and M_2 be arbitrary n-tape automata, for some $n > 0$. Let M'_1 be the automaton constructed from M_1 as in Lemma 6.3. Let M''_2 be the automaton constructed from M_2 as in Lemma 6.4. Then $L(M_1) \subseteq L(M_2)$ if and only if M'_1 and M''_2 are weakly equivalent.

□

Corollary 6.5:

The weak equivalence problem for independent location iteration schemas is unsolvable.

Proof: Immediate from Theorem 6.1, Lemma 6.2, and Theorem 6.2.

□

6.6 Single State Automata.

The problems which prevent the strong simulation of multi-tape automata by iteration schemas stem from the fact that an interpretation for a simulating schema must include a suitable "definition" of the sequence of states entered by the automata during its computation; the schema's execution must diverge if this sequence is incorrect or is unreasonably defined. Since equivalent automata will generally have different state sets, there is no way to ensure that schemas which simulate equivalent automata will both diverge or both converge under a given pair of consistent interpretations. In particular, we can generally provide such schemas with interpretations which are reasonable (in the sense of Section 6.4.1) for one schema but unreasonable for the other, thus forcing one schema to diverge under its interpretation while permitting the other to converge.

We would expect, on the other hand, that automata with a single state would present no such problems, although we might question whether such automata are capable of recognizing any interesting languages and whether their equivalence problems are related in any non-trivial way to those for multi-tape automata in general.

In this section we demonstrate that the iteration schemas are indeed capable of strongly simulating such automata, and also show that the automata are capable of recognizing non-trivial languages. We show, in fact, that the equivalence problem for multi-tape automata reduces to a restricted equivalence problem for the single state automata.

6.6.1 Single State Automata — Definitions.

Actually, the term single state automaton is something of a misnomer, since the automata which we define below contain two states. The function of the initial state of an automaton, however, is simply to ensure that the automaton begin a computation in one of some finite number of designated configurations. In particular, once the initial state is left no transitions back into the state are permitted; also, no movement of the tape heads is permitted while the automaton is in its initial state or moving out of the state. (Intuitively, we might think of the initial state as something of an "input monitor": if the tuple of initial symbols on the automaton's tapes is acceptable, the control state of the automaton is entered and the computation carried out. If the tuple is not acceptable, the initial state is never left and the computation diverges.)

The definition which follows is essentially the same as that given in Section 6.1 for multi-tape automata, except that we dispense with accepting and rejecting states and define the advancement and transition functions in such a way as to ensure that the initial state is as described above. (The automata will have no rejecting states, and their control states will function as accepting states.)

A <u>single state n-tape automaton</u> is a five-tuple

$$M = (T, q, q_I, f, h)$$

where: T is a finite set of <u>tape symbols</u>, including the endmarker \$.

q is the <u>control state</u> of the automaton.

q_I is the initial state of the automaton.

$f: \{q,q_I\} \times T^n \to \{q,q_I\}$ is the state transition function, a total function satisfying the property that $f(q,\varphi) = q$, all $\varphi \in T^n$.

$h: \{q,q_I\} \times T^n \to 2^{\{1,2,\ldots,n\}}$ is the head advancement function, a total function satisfying the properties that $h(q_I,\varphi) = \emptyset$, and $i \notin h(q,\varphi)$ whenever the ith component of φ is \$, for all $\varphi \in T^n$.

Configurations and computations are defined for single state automata in the same manner as for multi-tape automata in general. An n-tuple Λ of strings in T^* is accepted by M if there is a finite computation by M ending with the configuration $(q,\Lambda\cdot\$^n)$, and is rejected by M if no such computation exists. The languages accepted and rejected by M are defined as for multi-tape automata.

Strong and restricted equivalence are defined for single state automata as in Section 6.2. We shall not consider notions of weak equivalence for the automata.

6.6.2 Strong Equivalence — A Reducibility.

Let $M = (\{s_1,\ldots,s_k,\$\}, q, q_I, f, h)$ be a single state n-tape automaton, for some $n > 0$.

We shall show how to construct an independent location iteration schema S_M which simulates M. (The construction is quite similar to that described in Section 6.4.1, and we use much of the same terminology.)

S_M has variable symbols $x_1,\ldots,x_n$ and y. It has predicate symbols $p_{s_1},\ldots,p_{s_k},p_\$$ and p_m. It has a single function symbol g.

Expressions of the form $P_{s_i}(x_j)$ and $P_{\$}(x_j)$ are as defined in Section 6.4.1; we use the same enumeration of length n strings over the alphabet of M, and the expressions $P_{\varphi_1}, \ldots, P_{\varphi_{(k+1)^n}}$ are as defined in that section.

For each i, $1 \leq i \leq (k+1)^n$, we define S_i to be the schema:

$x_{h(q, \varphi_i)} \leftarrow g(x_{h(q,\varphi_i)})$

IF $p_m(y)$ THEN S_E ELSE $S_\uparrow$
$y \leftarrow g(y)$
.
.
.
IF $p_m(y)$ THEN S_E ELSE $S_\uparrow$
$y \leftarrow g(y)$

} ℓ times, where ℓ is the cardinality of $h_{(q,\varphi_i)}$

For each i, $1 \leq i < (k+1)^n$, we define R_i to be the schema:

IF P_{φ_i} THEN S_i ELSE R_{i+1}

We define $R_{(k+1)^n}$ to be the schema:

IF $P_{\varphi_{(k+1)^n}}$ THEN $S_{(k+1)^n}$ ELSE $S_\uparrow$

Now, let $\delta_1, \ldots, \delta_\ell$ be an enumeration of those length n strings for which $f(q_I, \delta_i) = q$, $1 \leq i \leq \ell$. Then S_M is the schema:

IF $(P_{\delta_1} \vee \ldots \vee P_{\delta_\ell})$ THEN S_E ELSE $S_\uparrow$
WHILE $p_m(y)$ DO R_1
IF $(P_{\$}(x_1) \wedge \ldots \wedge P_{\$}(x_n))$ THEN S_E ELSE $S_\uparrow$

We define reasonable interpretations for S_M in a manner analogous to that in Section 6.4.1, and note that each reasonable interpretation defines an n-tuple of strings over M's alphabet. The reader may verify that S_M diverges for all unreasonable interpretations, and converges for a reasonable interpretation I if and only if I defines a tuple of strings accepted by M and assigns to p_m a predicate Π_{p_m} such that:

$$\Pi_{p_m}(g^i \cdot \Delta_y) = \underline{\text{true}}, \text{ all } i < \ell$$
$$\Pi_{p_m}(g^i \cdot \Delta_y) = \underline{\text{false}}, \; i = \ell$$

where ℓ is the number of symbols in the tuple of strings defined by I. In such a case, the final value associated with y will be $g^{\ell} \cdot \Delta_y$.

We have immediately:

Theorem 6.3:

The strong equivalence problem for single state multi-tape automata reduces to that for independent location iteration schemas.

Proof: Let M and M' be single state n-tape automata, for some $n > 0$. We construct from M and M' the simulating schemas S_M and $S_{M'}$ as above, and note that M and M' are strongly equivalent if and only if S_M and $S_{M'}$ are strongly equivalent.

□

6.7 Equivalence of Multi-Tape Automata — A Reducibility Result.

We have shown that the strong equivalence problem for single state multi-tape automata reduces to that for structured independent location schemas. In this section, we provide motivation for that result by showing that such automata constitute a surprisingly rich class of multi-tape automata, and are in fact capable of simulating arbitrary automata if we allow them additional tapes on which to store control information. Since the particular information which is stored on these tapes will depend on the automaton being simulated, we must content ourselves with showing that the strong equivalence problem for multi-tape automata reduces to a restricted equivalence problem for the single state automata.

We begin with some useful definitions:

Let $\varphi = s_{i_1} \cdot s_{i_2} \cdot \ldots \cdot s_{i_k} \cdot s_{i_{k+1}} \cdot \ldots$ be a string over some alphabet T which does not contain the special symbol #. Then an <u>expansion</u> of φ is any string of the form:

$$\varphi_e = s_{i_1} \cdot \delta_1 \cdot (\#)^{j_1} \cdot s_{i_2} \cdot \delta_2 \cdot (\#)^{j_2} \cdot \ldots \cdot s_{i_k} \cdot \delta_k \cdot (\#)^{j_k} \cdot s_{i_{k+1}} \cdot \delta_{k+1} \cdot (\#)^{j_{k+1}} \ldots$$

in which for each i, δ_i is an arbitrary string in T^* and j_i is an integer greater than 0. We extend the notion of expansion to tuples of strings and sets of such tuples in the obvious manner:

If $\Lambda = (\varphi_1, \ldots, \varphi_n)$ is a tuple of strings over T, then an <u>expansion</u> of Λ is any tuple $\Lambda_e = (\varphi_{1_e}, \ldots, \varphi_{n_e})$ in which for each i, $1 \leq i \leq n$, φ_{i_e} is an expansion of φ_i. If X is a set of tuples of strings over T, then the <u>expansion</u> of X is the set $EXP(X) = \{\Lambda_e \mid \Lambda_e$ is an expansion of some $\Lambda \in X\}$.

Let M be an n-tape automaton with tape alphabet T and control states represented by a set of symbols $\hat{Q}$, and such that no more than a single tape head is advanced during any step in a computation by M. Let $\Lambda = \{\varphi_1, \ldots, \varphi_n\}$ be a tuple of strings accepted by M. Then the trace of Λ with respect to M is the n+4-tuple of strings $\Lambda_t = \{\alpha, \beta, \gamma, \gamma, \varphi_1, \ldots, \varphi_n\}$ over $T \cup \hat{Q} \cup \{\bar{1}, \ldots, \bar{n}\}$, such that $\alpha \in \{\bar{1}, \ldots, \bar{n}\}^*$ represents the sequence of tape heads moved during the computation by which M accepts Λ, $\beta \in T^*$ is the sequence of symbols scanned during the computation, and $\gamma \in \hat{Q}^*$ represents the sequence of states entered by M during the computation. We denote by TRACE(M) the set $\{\Lambda_t \mid \Lambda_t$ is the trace of some Λ accepted by M$\}$.

As we shall see, it is precisely the information in the first four components of a trace which constitutes the control information required by a single state automaton in the simulation of an arbitrary multi-tape automaton.

We note that if φ is a string over an alphabet T not containing #, then the expansion of φ formed by inserting # between each pair of symbols in φ cannot be obtained as an expansion of any other string. Thus we have:

Lemma 6.6:

Let M and M' be n-tape finite automata whose tape alphabets do not contain the symbol #. Then M and M' accept the same language if and only if EXP(L(M)) = EXP(L(M')). ⌟

Lemma 6.7:

Let M be an n-tape automaton $(T, Q, q_I, q_a, q_r, f, h)$ such that exactly one tape head is advanced during each step of any convergent computation by M, some $n > 0$. Then we may construct from M a single-state automaton M' with $n+4$ tapes, such that $L(M') = EXP(TRACE(M))$.

Proof: A formal definition of M' appears in Appendix C. We describe the behavior of M' informally below:

Let $\hat{Q}$ be the set of symbols $\{\hat{q}_i \mid q_i \text{ is a state in } Q\}$, and let $N = \{\overline{1}, \overline{2}, \ldots, \overline{n}\}$. The alphabet of M' is the set $T' = T \cup \hat{Q} \cup \overline{N} \cup \{\#\}$, where # is a special symbol not appearing in T.

We may represent a configuration of M' as an $n+4$-tuple of strings over T', and we say that a configuration Λ is a base configuration of M' if TAIL(Λ) is of the form $\overline{\ell} \cdot s \cdot \hat{q}_i \cdot \hat{q}_i \cdot \varphi$, where $\overline{\ell} \in \overline{N}$ is such that $\ell = h(q_i, \varphi)$ in M, and s is the ℓth symbol in $\varphi \in T^n$. Intuitively, a base configuration of M' represents some configuration of M just prior to some step α in a computation by M. The first four tapes in such a base configuration contain the following information:

(1) The head of M which is to be advanced at step α of M's computation.

(2) The symbol currently under scan by this head.

(3) The current internal state of M, recorded on each of tapes 3 and 4.

We shall describe the sequence of moves by which M' advances

to a new base configuration representing the configuration of M at the conclusion of step α of its computation:

M' moves head $\ell+4$ until the symbol # is scanned. It then moves head 4 until # is scanned, and likewise head 1. (Intuitively, we may think of M' as preparing these tapes for its next base configuration.)

M' advances head 4 past any number of #'s until a non-# is scanned. Unless this symbol is $\hat{q}_j$, where $q_j = f(q_i,\varphi)$ in M, the computation diverges, i.e. M' advances no tape head. This behavior of M' ensures that tape 4 has the symbol representing the internal state of M after step α of its computation.

M' advances head 2 and then head 3 until #'s are scanned on these tapes. It then advances head $\ell+4$ past any number of #'s until some symbol $s' \in T$ is scanned on the tape, diverging if the first symbol after the #'s is <u>not</u> a symbol in T. At this point, M' has discarded the original scanned symbol and internal state of M since this information is no longer needed; it has also advanced the appropriate tape head and scanned symbol s', so that each of the last n tape heads is scanning a symbol in T. Let φ' denote the string composed of these n scanned symbols:

M' advances head 2 past #'s, until some $s'' \in T$ is scanned such that s'' is under scan on the $k+4$th tape of M', where k is $h(q_j,\varphi')$. If the first symbol after the #'s is not such an s'', M' diverges. M' will thus have ensured that the "symbol under scan" component of its next base configuration is correct.

M' now advances head 1 past any number of #'s until a symbol $\overline{\ell}' \in \overline{N}$ is scanned such that $\ell' = h(q_j, \varphi')$. If any symbol other than $\overline{\ell}'$ immediately follows the #'s, M' diverges. M' thus ensures that the "head to be advanced" component of its next base configuration is correct.

Finally, M' advances head 3 past any number of #'s until symbol $\hat{q}_j$ is scanned on that tape, diverging if a symbol other than $\hat{q}_j$ is scanned first. At this point, M' has reached the desired base configuration.

The reader may verify that the conditions governing moves are unique in each case, and thus that the behavior of M' as outlined is consistent with the requirement that M' have a single control state. (Of course, we need to add rules ensuring that a computation of M' begins at a base configuration representing an initial configuration of M, and additional rules allowing a computation of M' to terminate if a base configuration represents a terminal configuration of M, but the addition of such rules is straightforward: the curious reader is referred to Appendix C).

□

We are now in a position to prove the main result of this section:

Theorem 6.4:

The strong equivalence problem for multi-tape automata reduces to a restricted equivalence problem for single state automata.

Proof: Let M and M' be arbitrary n-tape automata, for some $n > 0$. From Lemma 6.1, we can assume without loss of generality that exactly one tape head moves during each step of any convergent computation by either schema. Let M_s and M'_s be single state acceptors for EXP(TRACE(M)) and EXP(TRACE(M')), respectively, constructed as in the preceding lemma. Then from Lemma 6.6, $L(M) = L(M')$ if and only if M_s and M'_s are N-restricted equivalent, $N = \{5,6,\ldots,n+4\}$. We may interchange the accepting and rejecting states of M and M' to obtain n-tape automata $\overline{M}$ and $\overline{M}'$ such that $L(\overline{M}) = \overline{L}(M)$ and $L(\overline{M}') = \overline{L}(M')$. If we construct single state acceptors $\overline{M}_s$ and $\overline{M}'_s$ for EXP(TRACE($\overline{M}$)) and EXP(TRACE($\overline{M}'$)), then M is equivalent to M' if and only if M_s and M'_s, and $\overline{M}_s$ and $\overline{M}'_s$, are pairwise N-restricted equivalent.

□

6.7 Discussion.

We have attempted in this chapter to relate the equivalence problems for structured and non-structured independent location schemas. We began by looking at a class of automata equivalent to the non-structured schemas, and showed that such automata could be simulated by structured schemas which diverge whenever they are provided with interpretations not corresponding to valid computations. While such a simulation might be of some practical interest, Corollary 6.5 shows that it is of little value if our concern is with such issues as schematological equivalence. In fact, the corollary suggests that weak equivalence is likely to be undecidable even for classes of schemas for which strong equivalence is decidable. If we are looking for potentially solvable problems or potentially useful reducibilities, therefore, we had best restrict attention to strong equivalence and strong simulations.

Via Theorems 6.3 and 6.4, we have been able to establish another relation between the equivalence problems for the structured and non-structured schemas. While the relation is admittedly somewhat indirect, it never-the-less raises some questions (for example, regarding the decidability of equivalence for single state automata) which are of interest in their own right. It would seem, therefore, that this relation is worth pursuing.

CHAPTER VII: SUMMARY

In this thesis we have introduced iteration schemas (monadic schemas composed of assignment statements, conditional statements, and iteration statements) and have shown that such schemas correspond to monadic program schemas with structured flowcharts. We have also shown that the schemas form an incomplete subclass of the monadic program schemas in the sense that there exist monadic program schemas which are not equivalent to any iteration schema.

We have defined several subclasses of iteration schemas:

(i) Free schemas in which tests are never repeated during schema executions.

(ii) Single appearance schemas in which predicate symbols occur only once.

(iii) Conditional-free schemas composed solely of assignment statements and iteration statements.

(iv) Independent location schemas in which the assignment and argument variables are one and the same in each assignment statement.

(v) Restricted location schemas in which initial assignment statements of the form 'x ← y' are permitted, but which otherwise are independent location schemas.

We have formalized the notion of schema equivalence as the functional equivalence of schemas under free interpretations, and have explored the equivalence problems for these various classes of iteration schemas. We

were able to show that equivalence is undecidable for restricted location schemas, and strengthened the proof to demonstrate the undecidability of equivalence for such schemas which are conditional-free. We concluded that restricting the schemas to a single function and predicate symbol avails nothing in solvability, and that equivalence is unsolvable for such schemas with just two variable symbols. We have also shown that arbitrary nesting of iteration statements is not essential to unsolvable problems, and that unsolvability is with us for schemas with just two levels of iteration and for conditional-free schemas with just three levels.

We considered other features of the schemas for which we were able to demonstrate unsolvability, and this suggested several possibly solvable domains. We established techniques enabling us to demonstrate the decidability of equivalence for free schemas which are single appearance, or which are independent location and conditional-free; and for the class of schemas which are single appearance and conditional-free. While several interesting problems are left open, we are confident that the methodology developed can be extended to more general classes of iteration schemas. The results established in this thesis are summarized in Fig.10, along with some of the more interesting open problems.

We have explored in some detail the way in which equivalence problems for independent location iteration schemas are related to equivalence problems for independent location program schemas and multi-tape finite automata. We have shown that the iteration schemas are capable of "weakly" simulating such automata, and have used this result to demonstrate

FREE	SINGLE APPEARANCE	CONDITIONAL FREE	RESTRICTED LOCATION	INDEPENDENT LOCATION	
			X		Unsolvable
		X	X		Unsolvable
X	X				Solvable
X		X	X		Solvable
	X	X			Solvable
	X				Solvable*
X					Open
				X	Open
X				X	Open
		X		X	Open
X		X			Open
X		X	X		Open
Multi-Tape Finite Automata					Open
Single State Multi-Tape Automata					Open

Directed arcs indicate reducibilities: A ⟶ B indicates that the solvability of A implies that of B. Trivial reducibilities are not shown.

*Solvable, but details are not given.

FIGURE 10: Summary of Results.

the undecidability of weak equivalence (i.e. equivalence for terminating computations) for the independent location iteration schemas. We have shown that the equivalence problem for multi-tape automata with a single control state is reducible to that for independent location iteration schemas, and have also demonstrated that the equivalence problem for the automata in general reduces to the problem of deciding whether single state automata are equivalent over some arbitrary subset of their tapes.

The obvious areas for further study are suggested by Fig.10. A positive solution to the equivalence problem for free iteration schemas would be a major achievement since it would lend solid support to the conjecture that non-freeness is an essential characteristic of unsolvable problems. Such a solution for restricted location schemas would also be welcome since it would at least establish a "freeness" boundary between decidability and undecidability for these schemas. Finally, a positive solution to the equivalence problem for independent location iteration schemas would be, we feel, a major step in the solution of the equivalence problem for independent location schemas in general; at the very least it would provide, by implying the decidability of equivalence for single state automata, an additional avenue by which to approach the equivalence problem for multi-tape automata in general, an open problem of long standing.

BIBLIOGRAPHY

1. Ashcroft, E., and Manna, Z.
"The translation of 'GoTo' Programs to 'While' Programs",
Information Processing 71, North-Holland Publishing Co., 1972.

2. Ashcroft, E., Manna, Z., and Pnueli, A.
"Decidable Properties of Monadic Functional Schemas",
International Symposium on the Theory of Machines and Computations (Israel), 1971.

3. Chandra, A.K.
"On the Properties and Applications of Program Schemas",
Ph.D. Thesis, Dept. of Computer Science, Stanford University, 1973.

4. deBakker, J., and Scott, D.
"A Theory of Programs"
Unpublished memo, 1969. Reported in [19].

5. Dennis, J.B., and Fosseen, J.
"Introduction to Data Flow Schemas",
Computation Structures Group Memo 81-1, MIT, 1973.

6. Ershov, A.P.
"Theory of Program Schemata",
CSG Document, Computation Structures Group, MIT, 1971.

7. Fosseen, J.B.
"Representation of Algorithms by Maximally Parallel Schemata",
S.M. Thesis, Dept. of Electrical Engineering, MIT, 1972.

8. Hopcroft, J.E., and Ullman, J.D.
"Formal Languages and Their Relation to Automata",
Addison-Wesley Publishing Co., 1969.

9. Ianov, Y.I.
"The Logical Scheme of Algorithms", English translation in Problems in Cybernetics, Vol.1, Pergamon Press, 1960.

10. Karp, R.M., and Miller, R.E.
"Properties of a Model for Parallel Computations",
SIAM J. Appl. Math., Vol. 14, No.6, 1966.

11. Keller, R.M.
"A Solvable Program Schema Equivalence Problem",
Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems, 1971.

12. Kfouri, D.
"Reducing the Decidability of Equivalence for Multi-Tape Automata to the Decidability of EOLT"
Private communication, 1974.

13. Leung, C.
(In progress)
M.S. Thesis, Department of Electrical Engineering, MIT, 1975.

14. Linderman, J.P.
"Productivity in Parallel Computation Schemata",
Report MAC TR-111, Project MAC, MIT, 1973.

15. Luckham, D., and Park, D.
"Undecidability of the Equivalence Problem for Program Schemata",
Bolt Beranek and Newman, Inc., Report 1141, 1964.

16. Luckham, D., Park, D., and Paterson, M.S.
"On Formalized Computer Programs",
J. of Computer and Systems Sciences, Vol.4, No.3, 1970.

17. Meyer, A., and Ritchie, D.
"The Complexity of Loop Programs",
Proc. 22nd National ACM Conference, 1967.

18. Paterson, M.S.
"Equivalence Problems in a Model of Computation",
Ph.D. Thesis, University of Cambridge, 1967.

19. Paterson, M.S.
"Decision Problems in Computational Models",
Proc. ACM Conference on Proving Assertions about Programs, 1972

20. Paterson, M.S., and Hewitt, C.
"Comparative Schematology"
Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, 1970.

21. Qualitz, J.E.
"Weakly Productive Computation Schemata",
S.M. Thesis, Dept. of Elictrical Engineering, MIT, 1972.

22. Qualitz, J.E.
"Decidability of Equivalence for a Class of Data Flow Schemas",
Tech. Memo 58, Project MAC, MIT, 1975.

23. Rabin, M., and Scott, D.
"Finite Automata and Their Decision Problems",
IBM J. of Research and Development, 3,2, 1959.

24. Rodriguez, J.
"A Graph Model for Parallel Computations",
Sc.D. Thesis, Department of Electrical Engineering, MIT, 1967.

25. Rosenburg, A.
"On Multi-Head Finite Automata",
IBM J. of Research and Development, 10,5, 1966.

26. Rutledge, J.
"On Ianov's Program Schemata",
JACM 11, 1, 1964.

27. Slutz, D.R.
"Flow Graph Schemata",
Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, 1970.

28. Tsichritzis, D.
"The Equivalence Problem of Simple Programs",
JACM 17,4, 1970.

APPENDIX A: Decidability of Freeness for Independent Location Schemas

Let S be an arbitrary independent location schema, and let F be its flowchart. We say that a pair of nodes in F are <u>similar</u> if both nodes are transfer nodes labelled 'p(x)' for some predicate symbol p and variable symbol x. We claim that S is free if and only if each directed path from a node n to a similar node n' in F contains an assignment node labelled 'x ← f(x)' where f is some function symbol and x is the variable symbol of n and n':

(IF) If such is the case, then no variable is ever tested twice with the same predicate during an execution of S without an intervening assignment to the variable. Since S is an independent location schema, it is clear that no test can ever be repeated during a free execution of S.

(ONLY IF) An essential characteristic of a free schema is that every path through its flowchart corresponds to an execution of the schema. If there exists a directed path from n to n' containing no assignment labelled 'x ← f(x)' for some f, then a value associated with x will be tested at both n and n' during any execution of S which traverses this path, and thus S cannot be free.

APPENDIX B: Undecidability of Equivalence for Restricted Location Schemas With One Function Symbol and One Predicate Symbol.

We shall redefine the schemas of Section 4.3. The schema S_0 is:

UNTIL p(x) DO S_E

The schema S_1 is:

WHILE p(x) DO S_E

Schemas S'_0 and S'_1 are defined similarly.

For each i, $1 \leq i \leq k$, A_{ω_i} is:

IF p(v) THEN $S_{s_{i_1}}$ ELSE S_f

$x \leftarrow f(x)$

$v \leftarrow f(v)$

IF p(v) THEN $S_{s_{i_2}}$ ELSE S_f

$x \leftarrow f(x)$

$v \leftarrow f(v)$

.
.
.

IF p(v) THEN $S_{s_{i_{\delta_i}}}$ ELSE S_f

$x \leftarrow f(x)$

$v \leftarrow f(v)$

where $s_{i_1} \cdot s_{i_2} \cdot \ldots \cdot s_{i_{\delta_i}} = \omega_i$, as before. Schemas of the form B_{γ_i} are defined similarly, with symbol u in place of v.

For each i, $1 < i < k$, R_i is:

$w_i \leftarrow f(w_i)$

IF $p(w_i)$ THEN A_{ω_i} ELSE S_E

IF $p(w_i)$ THEN B_{γ_i} ELSE R_{i+1}

The schema R_1 is:

$w_1 \leftarrow f(w_1)$

$z \leftarrow f(z)$

IF $p(w_1)$ THEN A_{ω_1} ELSE S_E

IF $p(w_1)$ THEN B_{γ_1} ELSE R_2

The schema R_k is:

$w_k \leftarrow f(w_k)$

IF $p(w_k)$ THEN A_{ω_k} ELSE S_E

IF $p(w_k)$ THEN B_{γ_k} ELSE $S_\uparrow$

The schema S is:

$y \leftarrow x$

$u \leftarrow v$

IF $p(z)$ THEN R_1 ELSE R_1

WHILE $p(z)$ DO R_1

IF $p(v)$ THEN $S_\uparrow$ ELSE S_E

IF $p(u)$ THEN $S_\uparrow$ ELSE S_E

We note here that, in contrast to the results established in [18] for monadic program schemas in general, we are unable to establish the undecidability result for schemas which have simultaneously single predicate and function symbols and two variable symbols.

APPENDIX C: Definition of M'.

Let T, $\bar{N}$, $\hat{Q}$, and T' be as in the proof of Lemma 6.7. In the following, s's shall denote elements of T, $\bar{\ell}$'s elements of $\bar{N}$, $\hat{q}$'s elements of $\hat{Q}$, lower case Greek letters elements of T^*, and x's arbitrary symbols in T' other than # and \$.

M' is (T', q, q_I', f', h') where f' is defined as follows:

i) $f'(q_I', X) = q$ if X is $\ell \cdot s \cdot \hat{q}_i \cdot \hat{q}_i \cdot \omega$, where q_i is $f(q_I, \omega)$ in M, $\ell = h(q_i, \omega)$ in M, and s is the ℓth symbol of ω.

ii) $f'(q, X) = q$, all $X \in (T')^{n+4}$.

and h' is defined as follows:

i) $h'(q, \bar{\ell} \cdot s \cdot \hat{q}_i \cdot \hat{q}_i \cdot \alpha \cdot x \cdot \beta) = \ell+4$, $|\alpha| = \ell-1$.

ii) $h'(q, \bar{\ell} \cdot s \cdot \hat{q}_i \cdot \hat{q}_i \cdot \alpha \cdot \# \cdot \beta) = 4$.

iii) $h'(q, x \cdot s \cdot \hat{q}_i \cdot \# \cdot \alpha \cdot \# \cdot \beta) = 1$.

iv) $h'(q, \# \cdot s \cdot \hat{q}_i \cdot \# \cdot \alpha \cdot \# \cdot \beta) = 4$.

v) $h'(q, \# \cdot s \cdot \hat{q}_i \cdot \hat{q}_j \cdot \alpha \cdot \# \cdot \beta) = 2$, if $q_j = h(q_i, \alpha \cdot s \cdot \beta)$ in M.

vi) $h'(q, \#\# \cdot \hat{q}_i \cdot \hat{q}_j \cdot \alpha \cdot \# \cdot \beta) = 3$.

vii) $h'(q, \#\#\# \cdot \hat{q}_j \cdot \alpha \cdot \# \cdot \beta) = |\alpha|+5$.

viii) $h'(q, \#\#\# \cdot \hat{q}_j \cdot \gamma) = 2$.

ix) $h'(q, \# \cdot s \cdot \# \cdot \hat{q}_j \cdot \gamma) = 1$, if $h(q_j, \gamma)$ is defined in M and equals ℓ for some ℓ such that s' is the ℓth symbol of γ; or s' = \$ and $\gamma = \$^n$.

x) $h'(q, \bar{\ell} \cdot s \cdot \# \cdot \hat{q}_j \cdot \gamma) = 3$ if $h(q_j \cdot \gamma) = \ell$ and s' is the ℓth symbol in γ.

xi) $h'(q, \#\$\cdot q_j \cdot q_j \cdot \$^n) = 1.$

xii) $h'(q, \$\$\cdot q_j \cdot q_j \cdot \$^n) = 3.$

xiii) $h'(q, \$\$\#\cdot q_j \cdot \$^n) = 4.$

xiv) $h'(q, \$\$\#\#\$^n) = 3.$

xv) $h'(q, \$\$\$\#\$^n) = 4.$

and is undefined in all other cases.

BIOGRAPHICAL NOTE

Joseph Edward Qualitz was born in Waltham, Massachusetts on April 1, 1948. He attended public schools in Waltham, and graduated from Waltham High School in 1966. Mr. Qualitz received an S.B. and S.M. in Electrical Engineering from MIT in 1972, and a Ph.D. in Computer Science in 1975.

As a graduate student at MIT, Mr. Qualitz served as a teaching assistant until June of 1972, at which time he became an instructor in the Department of Electrical Engineering. He was the recipient in 1973 of an Electrical Engineering Department Teaching Award, and remained an instructor until January of 1975, at which time he resigned the position and became a full time research assistant at Project MAC.

Mr. Qualitz expects to join the engineering department of Artisan Industries of Waltham.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 11 / 16 / 95

Report # LCS-TR152

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 150 (154-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☒ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAPS (1-150) UNNUMBERED TITLE PAGES, 2-149, UNNUMBERED BLANK
(151-154) SCANCONTROL TRGT'S (3)

Scanning Agent Signoff:
Date Received: 11/16/95 Date Scanned: 12/11/95 Date Returned: 12/11/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**